目錄

前言 1.1
修订记录 1.2
如何贡献 1.3
Docker 简介 1.4
- 什么是 Docker 1.4.1
- 为什么要用 Docker 1.4.2

基本概念 1.5
- 镜像 1.5.1
- 容器 1.5.2
- 仓库 1.5.3

安装 Docker 1.6
- Ubuntu 1.6.1
- Debian 1.6.2
- CentOS 1.6.3
- macOS 1.6.4
- Windows PC 1.6.5
- 镜像加速器 1.6.6

使用镜像 1.7
- 获取镜像 1.7.1
- 列出镜像 1.7.2
- 利用 commit 理解镜像构成 1.7.3
- 使用 Dockerfile 定制镜像 1.7.4
- Dockerfile 指令详解 1.7.5
  - COPY 复制文件 1.7.5.1
  - ADD 更高级的复制文件 1.7.5.2
  - CMD 容器启动命令 1.7.5.3
  - ENTRYPOINT 入口点 1.7.5.4

VISIT…

LANZAROTE Caliente.com

ENV 设置环境变量 1.7.5.5
ARG 构建参数 1.7.5.6
VOLUME 定义匿名卷 1.7.5.7
EXPOSE 暴露端口 1.7.5.8
WORKDIR 指定工作目录 1.7.5.9
USER 指定当前用户 1.7.5.10
HEALTHCHECK 健康检查 1.7.5.11
ONBUILD 为他人作嫁衣裳 1.7.5.12
参考文档 1.7.5.13
其它制作镜像的方式 1.7.6
删除本地镜像 1.7.7
实现原理 1.7.8
操作容器 1.8
启动 1.8.1
守护态运行 1.8.2
终止 1.8.3
进入容器 1.8.4
导出和导入 1.8.5
删除 1.8.6
访问仓库 1.9
Docker Hub 1.9.1
私有仓库 1.9.2
配置文件 1.9.3
数据管理 1.10
数据卷 1.10.1
数据卷容器 1.10.2
备份、恢复、迁移数据卷 1.10.3
使用网络 1.11
外部访问容器 1.11.1
容器互联 1.11.2

高级网络配置 1.12
快速配置指南 1.12.1
配置 DNS 1.12.2
容器访问控制 1.12.3
端口映射实现 1.12.4
配置 docker0 网桥 1.12.5
自定义网桥 1.12.6
工具和示例 1.12.7
编辑网络配置文件 1.12.8
实例：创建一个点到点连接 1.12.9
安全 1.13
内核命名空间 1.13.1
控制组 1.13.2
服务端防护 1.13.3
内核能力机制 1.13.4
其它安全特性 1.13.5
总结 1.13.6
底层实现 1.14
基本架构 1.14.1
命名空间 1.14.2
控制组 1.14.3
联合文件系统 1.14.4
容器格式 1.14.5
网络 1.14.6
Docker 三剑客之 Compose 项目 1.15
简介 1.15.1
安装与卸载 1.15.2
使用 1.15.3
命令说明 1.15.4
YAML 模板文件 1.15.5

实战 Django 1.15.6
实战 Rails 1.15.7
实战 wordpress 1.15.8
Docker 三剑客之 Machine 项目 1.16
简介 1.16.1
安装 1.16.2
使用 1.16.3
Docker 三剑客之 Docker Swarm 1.17
Swarm 简介 1.17.1
安装 Swarm 1.17.2
使用 Swarm 1.17.3
使用其它服务发现后端 1.17.4
Swarm 中的调度器 1.17.5
Swarm 中的过滤器 1.17.6
本章小结 1.17.7
Etcd 项目 1.18
简介 1.18.1
安装 1.18.2
使用 etcdctl 1.18.3
CoreOS 项目 1.19
简介 1.19.1
工具 1.19.2
快速搭建CoreOS集群 1.19.3
Kubernetes 项目 1.20
简介 1.20.1
快速上手 1.20.2
基本概念 1.20.3
kubectl 使用 1.20.4
架构设计 1.20.5
Mesos - 优秀的集群资源调度平台 1.21

Mesos 简介 1.21.1
安装与使用 1.21.2
原理与架构 1.21.3
Mesos 配置项解析 1.21.4
日志与监控 1.21.5
常见应用框架 1.21.6
本章小结 1.21.7
容器与云计算 1.22
简介 1.22.1
亚马逊云 1.22.2
腾讯云 1.22.3
阿里云 1.22.4
小结 1.22.5
实战案例-操作系统 1.23
Busybox 1.23.1
Alpine 1.23.2
Debian\/Ubuntu 1.23.3
CentOS\/Fedora 1.23.4
CoreOS 1.23.5
本章小结 1.23.6
附录 1.24
附录一：常见问题总结 1.24.1
附录二：热门镜像介绍 1.24.2
Ubuntu 1.24.2.1
CentOS 1.24.2.2
MySQL 1.24.2.3
MongoDB 1.24.2.4
Redis 1.24.2.5
Nginx 1.24.2.6
WordPress 1.24.2.7

Node.js 1.24.2.8
附录三：Docker 命令查询 1.24.3
附录四：资源链接 1.24.4

Docker — 从入门到实践

0.8.3

Docker 是个划时代的开源项目，它彻底释放了计算虚拟化的威力，极大提高了应用的运行效率，降低了云计算资源供应的成本！使用 Docker，可以让应用的部署、测试和分发都变得前所未有的高效和轻松！

无论是应用开发者、运维人员、还是其他信息技术从业人员，都有必要认识和掌握 Docker，以在有限的时间内做更多有意义的事。

本开源书籍既适用于具备基础 Linux 知识的 Docker 初学者，也希望可供理解原理和实现的高级用户参考。同时，书中给出的实践案例，可供在进行实际部署时借鉴。前六章为基础内容，供用户理解 Docker 的基本概念和操作；7 ~ 9 章介绍一些高级操作；第 10 章给出典型的应用场景和实践案例；11、12 章介绍关于 Docker 安全和实现技术等高级话题。后续章节则分别介绍一些相关的热门开源项目。

在线阅读：GitBook 或 Github。

- pdf 版本 下载
- epub 版本 下载

Docker 自身仍在快速发展中，生态环境也在蓬勃成长。欢迎参与维护项目：如何贡献。

- 修订记录
- 贡献者名单

技术交流

欢迎关注 DockerPool 社区微博 @dockerpool，或加入 Docker 技术交流 QQ 群或微信组，分享 Docker 资源，交流 Docker 技术。

- QQ 群 I（已满）：341410255
- QQ 群 II（已满）：419042067
- QQ 群 III（已满）：210028779
- QQ 群 IV（已满）：483702734
- QQ 群 V（已满）：460598761

- QQ 群 VI （已满）：581983671
- QQ 群 VII （已满）：252403484
- QQ 群 VIII（已满）：544818750
- QQ 群 IX （可加）：571502246

进阶学习

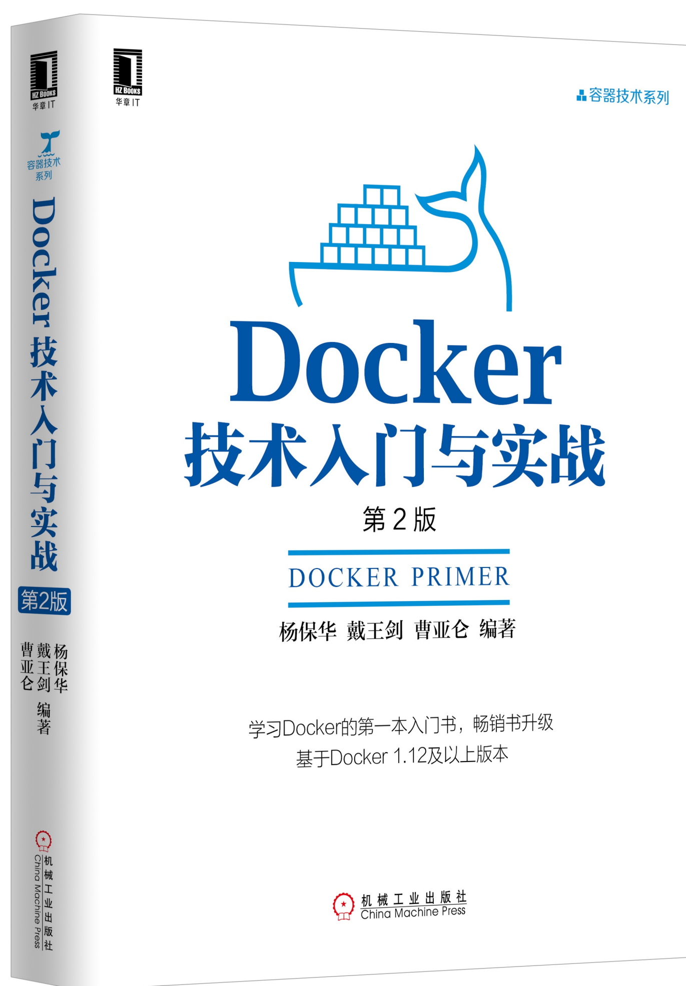


图 *1.1.1 - Docker* 技术入门与实战

《Docker 技术入门与实战》第二版已经正式出版，针对初版进行内容升级，欢迎大家阅读使用并反馈建议。

- 京东图书

- China-Pub

鼓励项目

欢迎鼓励项目一杯 coffee~



图 *1.1.2 - coffee*

主要修订记录

- 0.9.0: 2017-XX-YY
    - 完善 Compose 项目
    - 更新 Swarm 项目
    - 更新 Mesos 项目
    - 添加操作系统实战案例
- 0.8.0: 2017-01-08
    - 修正文字内容
    - 根据最新版本（1.12）修订安装使用
    - 补充附录章节
- 0.7.0: 2016-06-12
    - 根据最新版本进行命令调整
    - 修正若干文字描述
- 0.6.0: 2015-12-24
    - 补充 Machine 项目
    - 修正若干 bug
- 0.5.0: 2015-06-29
    - 添加 Compose 项目
    - 添加 Machine 项目
    - 添加 Swarm 项目
    - 完善 Kubernetes 项目内容
    - 添加 Mesos 项目内容
- 0.4.0: 2015-05-08
    - 添加 Etcd 项目
    - 添加 Fig 项目
    - 添加 CoreOS 项目
    - 添加 Kubernetes 项目
- 0.3.0: 2014-11-25
    - 完成仓库章节；
    - 重写安全章节；

  - 修正底层实现章节的架构、命名空间、控制组、文件系统、容器格式等内容；
  - 添加对常见仓库和镜像的介绍；
  - 添加 Dockerfile 的介绍；
  - 重新校订中英文混排格式。
  - 修订文字表达。
  - 发布繁体版本分支：zh-Hant。
- 0.2.0: 2014-09-18
  - 对照官方文档重写介绍、基本概念、安装、镜像、容器、仓库、数据管理、网络等章节；
  - 添加底层实现章节；
  - 添加命令查询和资源链接章节；
  - 其它修正。
- 0.1.0: 2014-09-05
  - 添加基本内容;
  - 修正错别字和表达不通顺的地方。

如何贡献项目

- 在 GitHub 上 `fork` 到自己的仓库，如 `docker_user/docker_practice` ，然后 `clone` 到本地，并设置用户信息。

```
$ git clone git@github.com:docker_user/docker_practice.git
$ cd docker_practice
$ git config user.name "yourname"
$ git config user.email "your email"
```

- 修改代码后提交，并推送到自己的仓库。

```
$ #do some change on the content
$ git commit -am "Fix issue #1: change helo to hello"
$ git push
```

- 在 GitHub 网站上提交 pull request。
- 定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/docker_practice
$ git fetch upstream
$ git checkout master
$ git rebase upstream/master
$ git push -f origin master
```

排版规范

本开源书籍遵循 中文排版指南 规范。

简介

本章将带领你进入 Docker 的世界。

什么是 Docker？

用它会带来什么样的好处？

好吧，让我们带着问题开始这神奇之旅。

什么是 Docker

Docker 最初是 dotCloud 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目，它是基于 dotCloud 公司多年云服务技术的一次革新，并于 2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在 GitHub 上进行维护。Docker 项目后来还加入了 Linux 基金会，并成立推动 开放容器联盟。

Docker 自开源后受到广泛的关注和讨论，至今其 GitHub 项目已经超过 3 万 6 千个星标和一万多个 fork。甚至由于 Docker 项目的火爆，在 2013 年底，dotCloud 公司决定改名为 Docker。Docker 最初是在 Ubuntu 12.04 上开发实现的；Red Hat 则从 RHEL 6.5 开始对 Docker 进行支持；Google 也在其 PaaS 产品中广泛应用 Docker。

Docker 使用 Google 公司推出的 Go 语言 进行开发实现，基于 Linux 内核的 cgroup，namespace，以及 AUFS 类的 Union FS 等技术，对进程进行封装隔离，属于 操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。最初实现是基于 LXC，从 0.7 以后开始去除 LXC，转而使用自行开发的 libcontainer，从 1.11 开始，则进一步演进为使用 runC 和 containerd。

Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护。使得 Docker 技术比虚拟机技术更为轻便、快捷。

下面的图片比较了 Docker 和传统虚拟化方式的不同之处。传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。

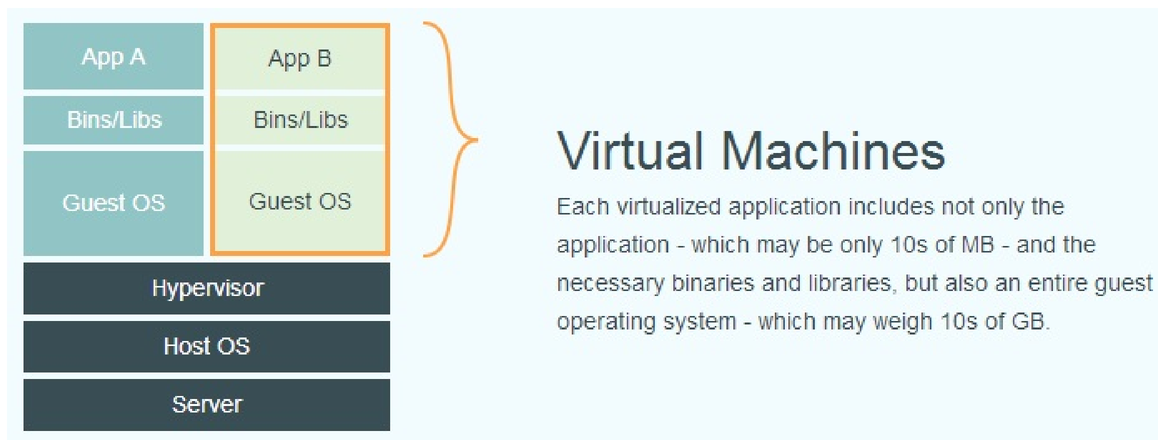


图 *1.4.1.1 - 传统虚拟化*

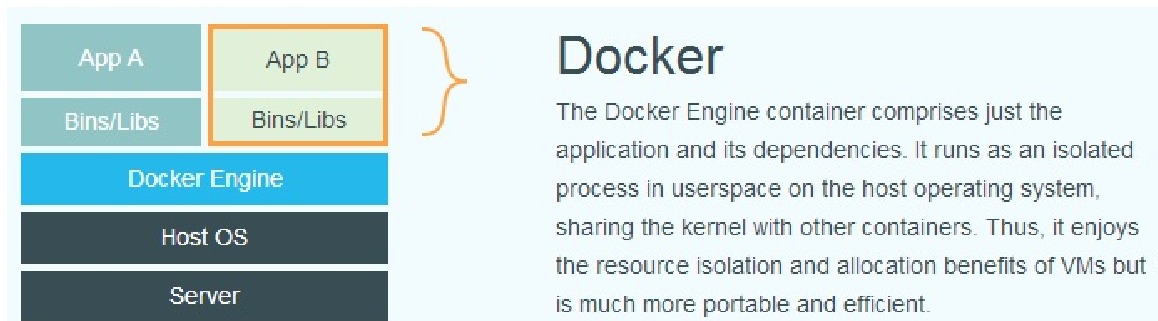


图 *1.4.1.2 - Docker*

为什么要使用 Docker？

作为一种新兴的虚拟化方式，Docker 跟传统的虚拟化方式相比具有众多的优势。

更高效的利用系统资源

由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，Docker 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

更快速的启动时间

传统的虚拟机技术启动应用服务往往需要数分钟，而 Docker 容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。

一致的运行环境

开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 bug 并未在开发过程中被发现。而 Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现 「这段代码在我机器上没问题啊」 这类问题。

持续交付和部署

对开发和运维（DevOps）人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。

使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。开发人员可以通过 Dockerfile 来进行镜像构建，并结合 持续集成(Continuous Integration) 系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜像，甚至结合 持续部署(Continuous Delivery/Deployment) 系统进行自动部署。

而且使用 `Dockerfile` 使镜像构建透明化，不仅仅开发团队可以理解应用运行环境，也方便运维团队理解应用运行所需条件，帮助更好的生产环境中部署该镜像。

更轻松的迁移

由于 Docker 确保了执行环境的一致性，使得应用的迁移更加容易。Docker 可以在很多平台上运行，无论是物理机、虚拟机、公有云、私有云，甚至是笔记本，其运行结果是一致的。因此用户可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

更轻松的维护和扩展

Docker 使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker 团队同各个开源项目团队一起维护了一大批高质量的官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

对比传统虚拟机总结

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

基本概念

Docker 包括三个基本概念

- 镜像（Image）
- 容器（Container）
- 仓库（Repository）

理解了这三个概念，就理解了 Docker 的整个生命周期。

Docker 镜像

我们都知道，操作系统分为内核和用户空间。对于 Linux 而言，内核启动后，会挂载 `root` 文件系统为其提供用户空间支持。而 Docker 镜像（Image），就相当于是一个 `root` 文件系统。比如官方镜像 `ubuntu:14.04` 就包含了完整的一套 Ubuntu 14.04 最小系统的 `root` 文件系统。

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

分层存储

因为镜像包含操作系统完整的 `root` 文件系统，其体积往往是庞大的，因此在 Docker 设计时，就充分利用 Union FS 的技术，将其设计为分层存储的架构。所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

关于镜像构建，将会在后续相关章节中做进一步的讲解。

Docker 容器

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的 `类` 和 `实例` 一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的 命名空间。因此容器可以拥有自己的 `root` 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。也因为这种隔离的特性，很多人初学 Docker 时常常会把容器和虚拟机搞混。

前面讲过镜像使用的是分层存储，容器也是如此。每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层，我们可以称这个为容器运行时读写而准备的存储层为容器存储层。

容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。

按照 Docker 最佳实践的要求，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用 数据卷（Volume）、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主(或网络存储)发生读写，其性能和稳定性更高。

数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器可以随意删除、重新 `run` ，数据却不会丢失。

Docker Registry

镜像构建完成后，可以很容易的在当前宿主上运行，但是，如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。

一个 **Docker Registry** 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。

通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 `<仓库名>:<标签>` 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 `latest` 作为默认标签。

以 Ubuntu 镜像 为例， `ubuntu` 是仓库的名字，其内包含有不同的版本标签，如， `14.04` , `16.04` 。我们可以通过 `ubuntu:14.04` ，或者 `ubuntu:16.04` 来具体指定所需哪个版本的镜像。如果忽略了标签，比如 `ubuntu` ，那将视为 `ubuntu:latest` 。

仓库名经常以 两段式路径 形式出现，比如 `jwilder/nginx-proxy` ，前者往往意味着 Docker Registry 多用户环境下的用户名，后者则往往是对应的软件名。但这并非绝对，取决于所使用的具体 Docker Registry 的软件或服务。

Docker Registry 公开服务

Docker Registry 公开服务是开放给用户使用、允许用户管理镜像的 Registry 服务。一般这类公开服务允许用户免费上传、下载公开的镜像，并可能提供收费服务供用户管理私有镜像。

最常使用的 Registry 公开服务是官方的 Docker Hub，这也是默认的 Registry，并拥有大量的高质量的官方镜像。除此以外，还有 CoreOS 的 Quay.io，CoreOS 相关的镜像存储在这里；Google 的 Google Container Registry，Kubernetes 的镜像使用的就是这个服务。

由于某些原因，在国内访问这些服务可能会比较慢。国内的一些云服务商提供了针对 Docker Hub 的镜像服务（Registry Mirror），这些镜像服务被称为加速器。常见的有 阿里云加速器、DaoCloud 加速器、灵雀云加速器等。使用加速器会直接从国

内的地址下载 Docker Hub 的镜像，比直接从官方网站下载速度会提高很多。在后面的章节中会有进一步如何配置加速器的讲解。

国内也有一些云服务商提供类似于 Docker Hub 的公开服务。比如 时速云镜像仓库、网易云镜像服务、DaoCloud 镜像市场、阿里云镜像库等。

私有 Docker Registry

除了使用公开服务外，用户还可以在本地搭建私有 Docker Registry。Docker 官方提供了 Docker Registry 镜像，可以直接使用做为私有 Registry 服务。在后续的相关章节中，会有进一步的搭建私有 Registry 服务的讲解。

开源的 Docker Registry 镜像只提供了 Docker Registry API 的服务端实现，足以支持 `docker` 命令，不影响使用。但不包含图形界面，以及镜像维护、用户管理、访问控制等高级功能。在官方的商业化版本 Docker Trusted Registry 中，提供了这些高级功能。

除了官方的 Docker Registry 外，还有第三方软件实现了 Docker Registry API，甚至提供了用户界面以及一些高级功能。比如，VMWare Harbor 和 Sonatype Nexus。

安装 Docker

Docker 在 1.13 版本之后，从 2017 年的 3 月 1 日开始，版本命名规则变为如下：

项目	说明
版本格式	YY.MM
stable 版本	每个季度发行
edge 版本	每个月发行
当前 Docker CE 版本	17.07.0

同时 Docker 划分为 CE 和 EE。CE 版本即社区版（免费，支持周期三个月），EE 即企业版，强调安全，付费使用。

Docker CE 每月发布一个 edge 版本 (17.03, 17.04, 17.05...)，每三个月发布一个 stable 版本 (17.03, 17.06, 17.09...)，Docker EE 和 stable 版本号保持一致，但每个版本提供一年维护。

官方网站上有各种环境下的 安装指南，这里主要介绍 Docker CE 在 Linux 、Windows 10 (PC) 和 macOS 上的安装。

Ubuntu 安装 Docker CE

准备工作

系统要求

Docker CE 支持以下版本的 Ubuntu 操作系统：

- Zesty 17.04
- Xenial 16.04 (LTS)
- Trusty 14.04 (LTS)

Docker CE 可以安装在 64 位的 x86 平台或 ARM 平台上。Ubuntu 发行版中，LTS（Long-Term-Support）长期支持版本，会获得 5 年的升级维护支持，这样的版本会更稳定，因此在生产环境中推荐使用 LTS 版本。

卸载旧版本

旧版本的 Docker 称为 `docker` 或者 `docker-engine` ，使用以下命令卸载旧版本：

```
$ sudo apt-get remove docker docker-engine docker.io
```

Ubuntu 14.04 可选内核模块

从 Ubuntu 14.04 开始，一部分内核模块移到了可选内核模块包 (`linux-image-extra-*`) ，以减少内核软件包的体积。正常安装的系统应该会包含可选内核模块包，而一些裁剪后的系统可能会将其精简掉。 `AUFS` 内核驱动属于可选内核模块的一部分，作为推荐的 Docker 存储层驱动，一般建议安装可选内核模块包以使用 `AUFS` 。

如果系统没有安装可选内核模块的话，可以执行下面的命令来安装可选内核模块包：

```
$ sudo apt-get update

$ sudo apt-get install \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual
```

使用 APT 镜像源 安装

由于官方源使用 HTTPS 以确保软件下载过程中不被篡改。因此，我们首先需要添加使用 HTTPS 传输的软件包以及 CA 证书。

```
$ sudo apt-get update

$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

鉴于国内网络问题，强烈建议使用国内源，下面先介绍国内源的使用。

国内源

为了确认所下载软件包的合法性，需要添加软件源的 GPG 密钥。

```
$ curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -
```

然后，我们需要向 `source.list` 中添加 Docker 软件源

```
$ sudo add-apt-repository \
    "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
```

> 以上命令会添加稳定版本的 Docker CE APT 镜像源，如果需要最新版本的 Docker CE 请将 stable 改为 edge 或者 test。从 Docker 17.06 开始，edge test 版本的 APT 镜像源也会包含稳定版本的 Docker。

官方源

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
 apt-key add -

$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

安装 Docker CE

更新 apt 软件包缓存，并安装 `docker-ce` ：

```
$ sudo apt-get update

$ sudo apt-get install docker-ce
```

使用脚本自动安装

在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，Ubuntu 系统上可以使用这套脚本安装：

```
$ curl -fsSL get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh --mirror Aliyun
```

执行这个命令后，脚本就会自动的将一切准备工作做好，并且把 Docker 安装在系统中。

启动 Docker CE

```
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

Ubuntu 14.04 请使用以下命令启动：

```
$ sudo service docker start
```

建立 docker 用户组

默认情况下， `docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好地做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

建立 `docker` 组：

```
$ sudo groupadd docker
```

将当前用户加入 `docker` 组：

```
$ sudo usermod -aG docker $USER
```

镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速。

参考文档

- Docker 官方 Ubuntu 安装文档
- 阿里云 Docker CE 安装镜像帮助

Debian 安装 Docker CE

准备工作

系统要求

Docker CE 支持以下版本的 Debian 操作系统：

* Stretch 9
* Jessie 8 (LTS)
* Wheezy 7.7 (LTS)

Docker CE 可以安装在 64 位的 x86 平台或 ARM 平台上（如树莓派）。

卸载旧版本

旧版本的 Docker 称为 `docker` 或者 `docker-engine`，使用以下命令卸载旧版本：

```
$ sudo apt-get remove docker docker-engine docker.io
```

Debian 7 Wheezy

Debian 7 的内核默认为 3.2，为了满足 Docker CE 的需求，应该安装 `backports` 的内核。

使用 APT 镜像源 安装

由于官方源使用 HTTPS 以确保软件下载过程中不被篡改。因此，我们首先需要添加使用 HTTPS 传输的软件包以及 CA 证书。

Debian 8 Jessie 或者 Debian 9 Stretch 使用以下命令:

```
$ sudo apt-get update

$ sudo apt-get install \
     apt-transport-https \
     ca-certificates \
     curl \
     gnupg2 \
     lsb-release \
     software-properties-common
```

Debian 7 Wheezy 使用以下命令：

```
$ sudo apt-get update

$ sudo apt-get install \
     apt-transport-https \
     ca-certificates \
     curl \
     lsb-release \
     python-software-properties
```

鉴于国内网络问题，强烈建议使用国内源，下面先介绍国内源的使用。

国内源

为了确认所下载软件包的合法性，需要添加软件源的 GPG 密钥。

```
$ curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/debian/gpg | sudo apt-key add -
```

然后，我们需要向 `source.list` 中添加 Docker CE 软件源：

```
$ sudo add-apt-repository \
   "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/debian \
   $(lsb_release -cs) \
   stable"
```

> 以上命令会添加稳定版本的 Docker CE APT 镜像源，如果需要最新版本的 Docker CE 请将 stable 改为 edge 或者 test。从 Docker 17.06 开始，edge test 版本的 APT 镜像源也会包含稳定版本的 Docker CE。

官方源

```
$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo
 apt-key add -

$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/debian \
   $(lsb_release -cs) \
   stable"
```

Debian 7 需要进行额外的操作：

编辑 `/etc/apt/sources.list` 将 deb-src 一行删除或者使用 # 注释。

```
deb-src [arch=amd64] https://download.docker.com/linux/debian wh
eezy stable
```

安装 Docker CE

更新 apt 软件包缓存，并安装 `docker-ce` 。

```
$ sudo apt-get update

$ sudo apt-get install docker-ce
```

使用脚本自动安装

在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，Debian 系统上可以使用这套脚本安装：

```
$ curl -fsSL get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh --mirror Aliyun
```

执行这个命令后，脚本就会自动的将一切准备工作做好，并且把 Docker CE 的 edge 版本安装在系统中。

树莓派安装 Docker CE

基本和上述步骤相同，添加 apt 源时请使用如下命令：

```
$ sudo add-apt-repository \
    "deb [arch=armhf] https://mirrors.aliyun.com/docker-ce/linux
/debian \
    $(lsb_release -cs) \
    stable"
```

ARM 平台不能使用 x86 镜像，查看树莓派可使用镜像请访问 arm32v7。

启动 Docker CE

```
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

Debian 7 Wheezy 请使用以下命令启动

```
$ sudo service docker start
```

建立 docker 用户组

默认情况下， `docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好地做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

建立 `docker` 组：

```
$ sudo groupadd docker
```

将当前用户加入 `docker` 组：

```
$ sudo usermod -aG docker $USER
```

镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速。

参考文档

- Docker 官方 Debian 安装文档
- 阿里云 Docker CE 安装镜像帮助

CentOS 安装 Docker CE

准备工作

系统要求

Docker CE 支持 64 位版本 CentOS 7，并且要求内核版本不低于 3.10。 CentOS 7 满足最低内核的要求，但由于内核版本比较低，部分功能（如 `overlay2` 存储层驱动）无法使用，并且部分功能可能不太稳定。

卸载旧版本

旧版本的 Docker 称为 `docker` 或者 `docker-engine`，使用以下命令卸载旧版本：

```
$ sudo yum remove docker \
                  docker-common \
                  docker-selinux \
                  docker-engine
```

使用 yum 源 安装

执行以下命令安装依赖包：

```
$ sudo yum install -y yum-utils device-mapper-persistent-data lv
m2
```

鉴于国内网络问题，强烈建议使用国内源，下面先介绍国内源的使用。

国内源

执行下面的命令添加 `yum` 软件源：

```
$ sudo yum-config-manager \
    --add-repo \
    https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.
repo
```

> 以上命令会添加稳定版本的 Docker CE yum 源。从 Docker 17.06 开始，edge test 版本的 yum 源也会包含稳定版本的 Docker CE。

官方源

```
$ sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

如果需要最新版本的 Docker CE 请使用以下命令：

```
$ sudo yum-config-manager --enable docker-ce-edge
```

```
$ sudo yum-config-manager --enable docker-ce-test
```

安装 Docker CE

更新 `yum` 软件源缓存，并安装 `docker-ce` 。

```
$ sudo yum makecache fast
$ sudo yum install docker-ce
```

使用脚本自动安装

在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，CentOS 系统上可以使用这套脚本安装：

```
$ curl -fsSL get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh --mirror Aliyun
```

执行这个命令后，脚本就会自动的将一切准备工作做好，并且把 Docker CE 的 edge 版本安装在系统中。

启动 Docker CE

```
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

建立 docker 用户组

默认情况下， `docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好地做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

建立 `docker` 组：

```
$ sudo groupadd docker
```

将当前用户加入 `docker` 组：

```
$ sudo usermod -aG docker $USER
```

镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速。

添加内核参数

默认配置下，如果在 CentOS 使用 Docker CE 看到下面的这些警告信息：

```
WARNING: bridge-nf-call-iptables is disabled
WARNING: bridge-nf-call-ip6tables is disabled
```

请添加内核配置参数以启用这些功能。

```
$ sudo tee -a /etc/sysctl.conf <<-EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
```

然后重新加载 `sysctl.conf` 即可

```
$ sudo sysctl -p
```

参考文档

- Docker 官方 CentOS 安装文档。
- 阿里云 Docker CE 安装镜像帮助

macOS 安装 Docker

系统要求

Docker for Mac 要求系统最低为 macOS 10.10.3 Yosemite，或者 2010 年以后的 Mac 机型，准确说是带 Intel MMU 虚拟化的，最低 4GB 内存。

如果系统不满足需求，可以安装 Docker Toolbox。

安装

使用 Homebrew 安装

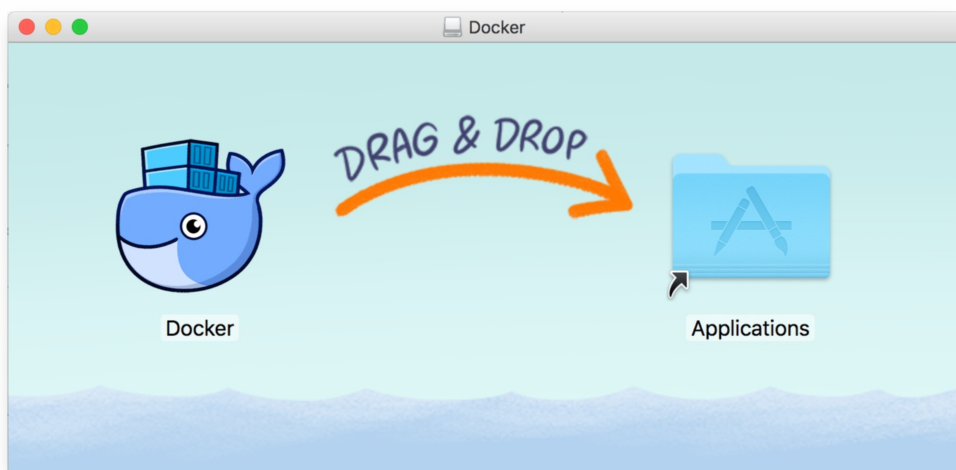
Homebrew 的 Cask 已经支持 Docker for Mac，因此可以很方便的使用 Homebrew Cask 来进行安装：

```
brew cask install docker
```

手动下载安装

如果需要手动下载，可以通过这个链接下载：https://download.docker.com/mac/stable/Docker.dmg

如同 macOS 其它软件一样，安装也非常简单，双击下载的 `.dmg` 文件，然后将那只叫 Moby 的鲸鱼图标拖拽到 `Application` 文件夹即可（其间可能会询问系统密码）。



运行

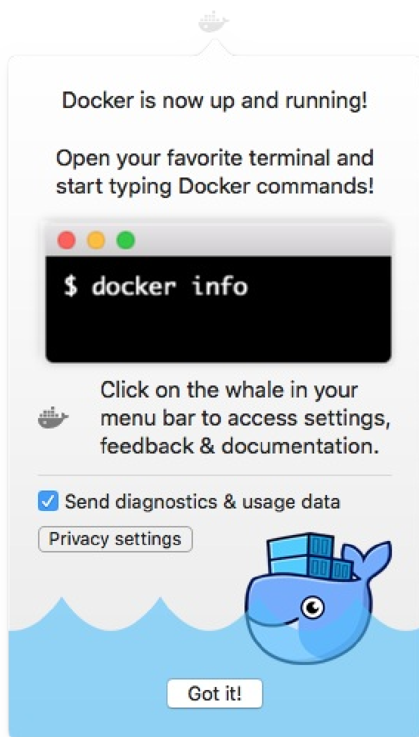
从应用中找到 Docker 图标并点击运行。



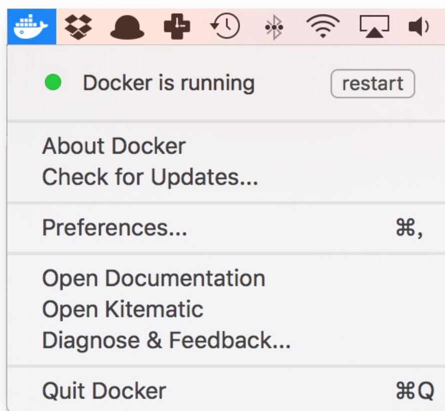
运行之后，会在右上角菜单栏看到多了一个鲸鱼图标，这个图标表明了 Docker 的运行状态。



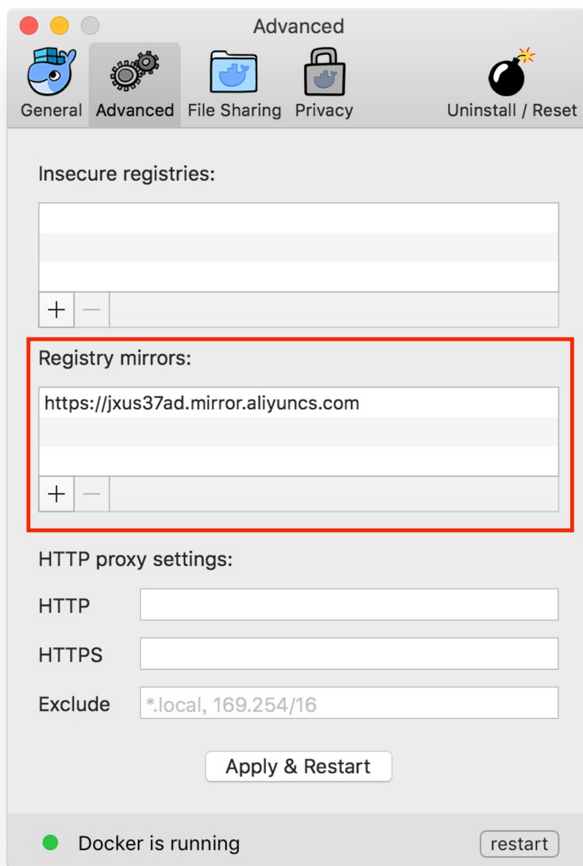
第一次点击图标，可能会看到这个安装成功的界面，点击 "Got it!" 可以关闭这个窗口。



以后每次点击鲸鱼图标会弹出操作菜单。



在国内使用 *Docker* 的话，需要配置加速器，在菜单中点击 `Preferences...` ，然后查看 `Advanced` 标签，在其中的 `Registry mirrors` 部分里可以点击加号来添加加速器地址。



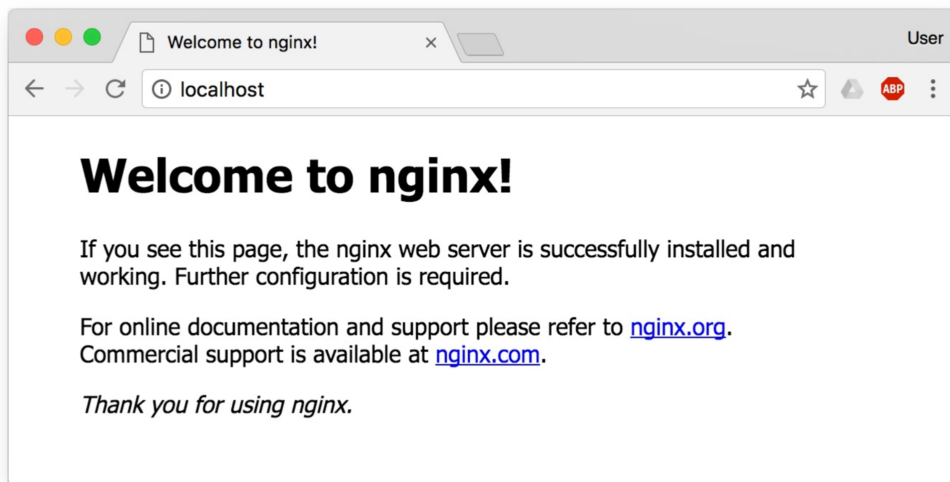
启动终端后，通过命令可以检查安装后的 Docker 版本。

```
$ docker --version
Docker version 1.12.3, build 6b644ec
$ docker-compose --version
docker-compose version 1.8.1, build 878cff1
$ docker-machine --version
docker-machine version 0.8.2, build e18a919
```

如果 `docker version` 、 `docker info` 都正常的话，可以运行一个 Nginx 服务器：

```
$ docker run -d -p 80:80 --name webserver nginx
```

服务运行后，可以访问 http://localhost，如果看到了 "Welcome to nginx!"，就说明 Docker for Mac 安装成功了。



要停止 Nginx 服务器并删除执行下面的命令：

```
$ docker stop webserver
$ docker rm webserver
```

参考文档

- Docker 官方 macOS 安装文档

Windows 10 PC 安装 Docker CE

系统要求

Docker CE 支持 64 位版本的 Windows 10 Pro，且必须开启 Hyper-V。

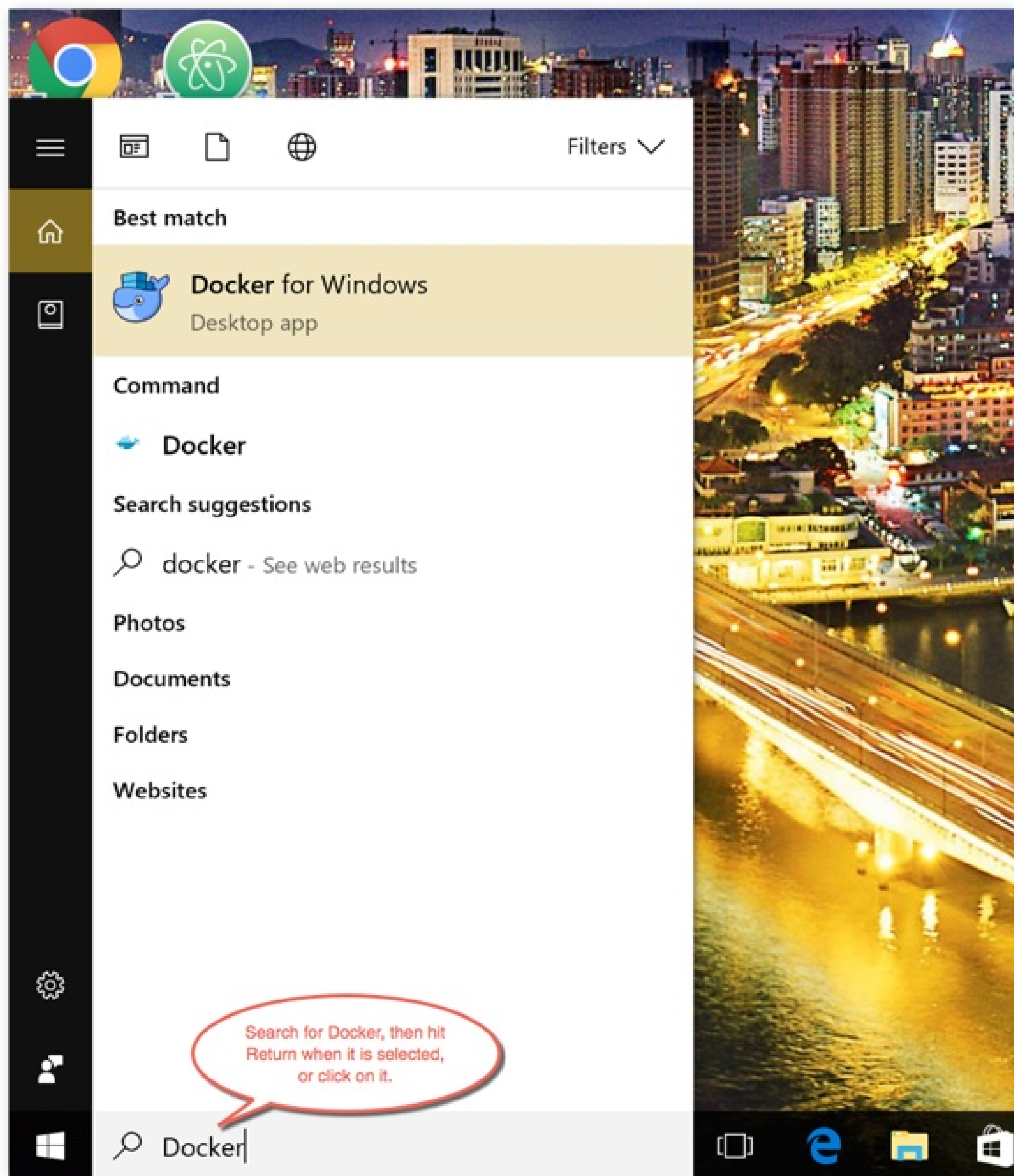
安装

点击以下链接下载 stable 或 edge 版本的 Docker CE。

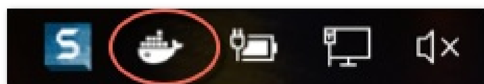
下载好之后双击 Docker for Windows Installer.exe 开始安装。

运行

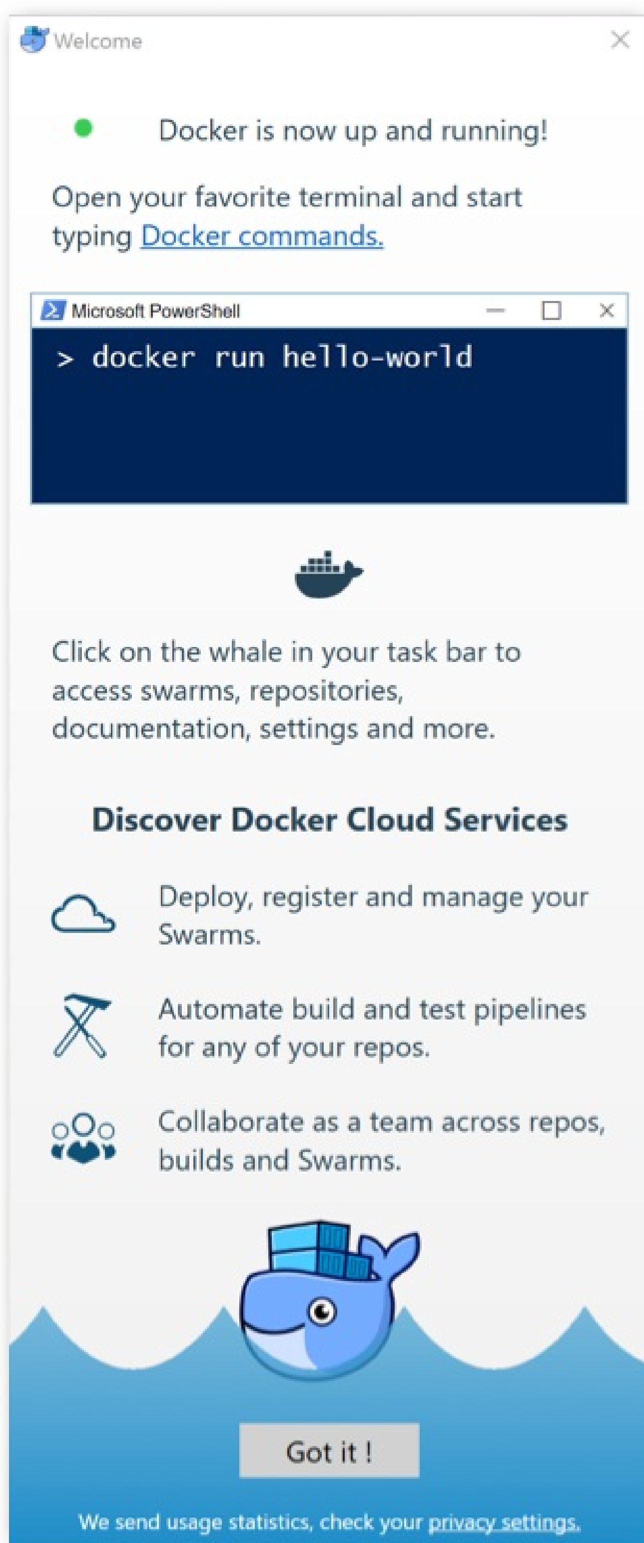
在 Windows 搜索栏 输入 Docker 点击 Docker for Windows 开始运行。



Docker CE 启动之后会在 Windows 任务栏出现鲸鱼图标。



等待片刻，点击 Got it 开始使用 Docker CE。



镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速 。

参考文档

- Docker 官方 Windows PC 安装文档

镜像加速器

国内访问 Docker Hub 有时会遇到困难，此时可以配置镜像加速器。Docker官方和国内很多云服务商都提供了加速器服务，例如：

- Docker 官方提供的中国registry mirror
- 阿里云加速器
- DaoCloud 加速器

注册用户并且申请加速器，会获得如 `https://jxus37ad.mirror.aliyuncs.com` 这样的地址。我们需要将其配置给 Docker 引擎。

Ubuntu 14.04、Debian 7 Wheezy

对于使用 upstart 的系统而言，编辑 `/etc/default/docker` 文件，在其中的 `DOCKER_OPTS` 中添加获得的加速器配置 `--registry-mirror=<加速器地址>`，如：

```
DOCKER_OPTS="--registry-mirror=https://jxus37ad.mirror.aliyuncs.
com"
```

重新启动服务。

```
$ sudo service docker restart
```

Ubuntu 16.04、Debian 8 Jessie、CentOS 7

对于使用 systemd 的系统，用 `systemctl enable docker` 启用服务后，编辑 `/etc/systemd/system/multi-user.target.wants/docker.service` 文件，找到 `ExecStart=` 这一行，在这行最后添加加速器地址 `--registry-mirror=<加速器地址>`，如：

```
ExecStart=/usr/bin/dockerd --registry-mirror=https://jxus37ad.mi
rror.aliyuncs.com
```

注：对于 *1.12* 以前的版本，`dockerd` 换成 `docker daemon` 。

重新加载配置并且重新启动。

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

Windows 10

对于使用 WINDOWS 10 的系统，在系统右下角托盘图标内右键菜单选择 `Settings` ，打开配置窗口后左侧导航菜单选择 `Docker Daemon` 。编辑窗口内的JSON串，填写如阿里云、DaoCloud之类的加速器地址，如：

```
{
  "registry-mirrors": [
    "https://sr5arhkn.mirror.aliyuncs.com",
    "http://14d216f4.m.daocloud.io"
  ],
  "insecure-registries": []
}
```

编辑完成，点击Apply保存后Docker服务会重新启动。

macOS

对于macOS的用户，如果你使用的是**Docker for Mac**，那配置起来很简单。在任务栏点击应用图标 -> Perferences... -> Daemon -> Registry mirrors。在列表中添加云服务商提供的加速器地址即可。修改完成之后，点击 `Apply & Restart` 按钮，Docker就会重启并应用配置的镜像地址了。

检查加速器是否生效

Linux系统下配置完加速器需要检查是否生效，在命令行执行 `ps -ef | grep dockerd` ，如果从结果中看到了配置的 `--registry-mirror` 参数说明配置成功。

```
$ sudo ps -ef | grep dockerd
root      5346     1  0 19:03 ?        00:00:00 /usr/bin/dockerd
 --registry-mirror=https://jxus37ad.mirror.aliyuncs.com
$
```

如果 `Docker` 版本大于1.13或17.05.0-ce，也可以

```
$ sudo docker info|grep "Registry Mirrors" -A 1
Registry Mirrors:
 https://registry.docker-cn.com/
```

使用 Docker 镜像

在之前的介绍中，我们知道镜像是 Docker 的三大组件之一。

Docker 运行容器前需要本地存在对应的镜像，如果镜像不存在本地，Docker 会从镜像仓库下载（默认是 Docker Hub 公共注册服务器中的仓库）。

本章将介绍更多关于镜像的内容，包括：

- 从仓库获取镜像；
- 管理本地主机上的镜像；
- 介绍镜像实现的基本原理。

获取镜像

之前提到过，Docker Hub 上有大量的高质量的镜像可以用，这里我们就说一下怎么获取这些镜像并运行。

从 Docker Registry 获取镜像的命令是 `docker pull` 。其命令格式为：

```
docker pull [选项] [Docker Registry地址]<仓库名>:<标签>
```

具体的选项可以通过 `docker pull --help` 命令看到，这里我们说一下镜像名称的格式。

- Docker Registry地址：地址的格式一般是 `<域名/IP>[:端口号]` 。默认地址是 Docker Hub。
- 仓库名：如之前所说，这里的仓库名是两段式名称，既 `<用户名>/<软件名>` 。对于 Docker Hub，如果不给出用户名，则默认为 `library` ，也就是官方镜像。

比如：

```
$ docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu
bf5d46315322: Pull complete
9f13e0ac480c: Pull complete
e8988b5b3097: Pull complete
40af181810e7: Pull complete
e6f7c7e5c03e: Pull complete
Digest: sha256:147913621d9cdea08853f6ba9116c2e27a3ceffecf3b49298
3ae97c3d643fbbe
Status: Downloaded newer image for ubuntu:14.04
```

上面的命令中没有给出 Docker Registry 地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 `ubuntu:14.04` ，因此将会获取官方镜像 `library/ubuntu` 仓库中标签为 `14.04` 的镜像。

从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储所构成。下载也是一层层的去下载，并非单一文件。下载过程中给出了每一层的 ID 的前 12 位。并且下载结束后，给出该镜像完整的 `sha256` 的摘要，以确保下载一致性。

在实验上面命令的时候，你可能会发现，你所看到的层 ID 以及 `sha256` 的摘要和这里的不一样。这是因为官方镜像是一直在维护的，有任何新的 bug，或者版本更新，都会进行修复再以原来的标签发布，这样可以确保任何使用这个标签的用户可以获得更安全、更稳定的镜像。

如果从 *Docker Hub* 下载镜像非常缓慢，可以参照 镜像加速器 一节配置加速器。

运行

有了镜像后，我们就可以以这个镜像为基础启动一个容器来运行。以上面的 `ubuntu:14.04` 为例，如果我们打算启动里面的 `bash` 并且进行交互式操作的话，可以执行下面的命令。

```
$ docker run -it --rm ubuntu:14.04 bash
root@e7009c6ce357:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.5 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.5 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
root@e7009c6ce357:/# exit
exit
$
```

`docker run` 就是运行容器的命令，具体格式我们会在后面的章节讲解，我们这里简要的说明一下上面用到的参数。

- `-it` ：这是两个参数，一个是 `-i` ：交互式操作，一个是 `-t` 终端。我们这里打算进入 `bash` 执行一些命令并查看返回结果，因此我们需要交互式终

端。

- `--rm` ：这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 `docker rm` 。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 `--rm` 可以避免浪费空间。
- `ubuntu:14.04` ：这是指用 `ubuntu:14.04` 镜像为基础来启动容器。
- `bash` ：放在镜像名后的是命令，这里我们希望有个交互式 Shell，因此用的是 `bash` 。

进入容器后，我们可以在 Shell 下操作，执行任何所需的命令。这里，我们执行了 `cat /etc/os-release` ，这是 Linux 常用的查看当前系统版本的命令，从返回的结果可以看到容器内是 `Ubuntu 14.04.5 LTS` 系统。

最后我们通过 `exit` 退出了这个容器。

列出镜像

要想列出已经下载下来的镜像，可以使用 `docker images` 命令。

```
$ docker images
REPOSITORY           TAG                 IMAGE ID            CRE
ATED             SIZE
redis                latest              5f515359c7f8        5 d
ays ago          183 MB
nginx                latest              05a60462f8ba        5 d
ays ago          181 MB
mongo                3.2                 fe9198c04d62        5 d
ays ago          342 MB
<none>               <none>              00285df0df87        5 d
ays ago          342 MB
ubuntu               16.04               f753707788c5        4 w
eeks ago         127 MB
ubuntu               latest              f753707788c5        4 w
eeks ago         127 MB
ubuntu               14.04               1e0c3dd64ccd        4 w
eeks ago         188 MB
```

列表包含了仓库名、标签、镜像 ID、创建时间以及所占用的空间。

其中仓库名、标签在之前的基础概念章节已经介绍过了。镜像 **ID** 则是镜像的唯一标识，一个镜像可以对应多个标签。因此，在上面的例子中，我们可以看到 `ubuntu:16.04` 和 `ubuntu:latest` 拥有相同的 ID，因为它们对应的是同一个镜像。

镜像体积

如果仔细观察，会注意到，这里标识的所占用空间和在 Docker Hub 上看到的镜像大小不同。比如，`ubuntu:16.04` 镜像大小，在这里是 `127 MB` ，但是在 Docker Hub 显示的却是 `50 MB` 。这是因为 Docker Hub 中显示的体积是压缩后的体积。在镜像下载和上传过程中镜像是保持着压缩状态的，因此 Docker Hub 所

显示的大小是网络传输中更关心的流量大小。而 `docker images` 显示的是镜像下载到本地后，展开的大小，准确说，是展开后的各层所占空间的总和，因为镜像到本地后，查看空间的时候，更关心的是本地磁盘空间占用的大小。

另外一个需要注意的问题是， `docker images` 列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于 Docker 镜像是多层存储结构，并且可以继承、复用，因此不同镜像可能会因为使用相同的基础镜像，从而拥有共同的层。由于 Docker 使用 Union FS，相同的层只需要保存一份即可，因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小的多。

虚悬镜像

上面的镜像列表中，还可以看到一个特殊的镜像，这个镜像既没有仓库名，也没有标签，均为 `<none>` 。：

```
<none>                <none>              00285df0df87        5 d
ays ago          342 MB
```

这个镜像原本是有镜像名和标签的，原来为 `mongo:3.2` ，随着官方镜像维护，发布了新版本后，重新 `docker pull mongo:3.2` 时， `mongo:3.2` 这个镜像名被转移到了新下载的镜像身上，而旧的镜像上的这个名称则被取消，从而成为了 `<none>` 。除了 `docker pull` 可能导致这种情况， `docker build` 也同样可以导致这种现象。由于新旧镜像同名，旧镜像名称被取消，从而出现仓库名、标签均为 `<none>` 的镜像。这类无标签镜像也被称为 虚悬镜像**(dangling image)** ，可以用下面的命令专门显示这类镜像：

```
$ docker images -f dangling=true
REPOSITORY          TAG                 IMAGE ID            CREA
TED              SIZE
<none>              <none>              00285df0df87        5 da
ys ago           342 MB
```

一般来说，虚悬镜像已经失去了存在的价值，是可以随意删除的，可以用下面的命令删除。

```
$ docker rmi $(docker images -q -f dangling=true)
```

中间层镜像

为了加速镜像构建、重复利用资源，Docker 会利用 中间层镜像。所以在使用一段时间后，可能会看到一些依赖的中间层镜像。默认的 `docker images` 列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像的话，需要加 `-a` 参数。

```
$ docker images -a
```

这样会看到很多无标签的镜像，与之前的虚悬镜像不同，这些无标签的镜像很多都是中间层镜像，是其它镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍，而这些镜像是别的镜像的依赖，因此并不会因为它们被列出来而多存了一份，无论如何你也会需要它们。只要删除那些依赖它们的镜像后，这些依赖的中间层镜像也会被连带删除。

列出部分镜像

不加任何参数的情况下，`docker images` 会列出所有顶级镜像，但是有时候我们只希望列出部分镜像。`docker images` 有好几个参数可以帮助做到这个事情。

根据仓库名列出镜像

```
$ docker images ubuntu
REPOSITORY          TAG                 IMAGE ID            CREA
TED             SIZE
ubuntu              16.04               f753707788c5        4 we
eks ago         127 MB
ubuntu              latest              f753707788c5        4 we
eks ago         127 MB
ubuntu              14.04               1e0c3dd64ccd        4 we
eks ago         188 MB
```

列出特定的某个镜像，也就是说指定仓库名和标签

```
$ docker images ubuntu:16.04
REPOSITORY          TAG                 IMAGE ID            CREA
TED             SIZE
ubuntu              16.04               f753707788c5        4 we
eks ago         127 MB
```

除此以外， `docker images` 还支持强大的过滤器参数 `--filter` ，或者简写 `-f` 。之前我们已经看到了使用过滤器来列出虚悬镜像的用法，它还有更多的用法。比如，我们希望看到在 `mongo:3.2` 之后建立的镜像，可以用下面的命令：

```
$ docker images -f since=mongo:3.2
REPOSITORY          TAG                 IMAGE ID            CREA
TED             SIZE
redis               latest              5f515359c7f8        5 da
ys ago          183 MB
nginx               latest              05a60462f8ba        5 da
ys ago          181 MB
```

想查看某个位置之前的镜像也可以，只需要把 `since` 换成 `before` 即可。

此外，如果镜像构建时，定义了 `LABEL` ，还可以通过 `LABEL` 来过滤。

```
$ docker images -f label=com.example.version=0.1
...
```

以特定格式显示

默认情况下， `docker images` 会输出一个完整的表格，但是我们并非所有时候都会需要这些内容。比如，刚才删除虚悬镜像的时候，我们需要利用 `docker images` 把所有的虚悬镜像的 ID 列出来，然后才可以交给 `docker rmi` 命令作为参数来删除指定的这些镜像，这个时候就用到了 `-q` 参数。

```
$ docker images -q
5f515359c7f8
05a60462f8ba
fe9198c04d62
00285df0df87
f753707788c5
f753707788c5
1e0c3dd64ccd
```

`--filter` 配合 `-q` 产生出指定范围的 ID 列表，然后送给另一个 `docker` 命令作为参数，从而针对这组实体成批的进行某种操作的做法在 Docker 命令行使用过程中非常常见，不仅仅是镜像，将来我们会在各个命令中看到这类搭配以完成很强大的功能。因此每次在文档看到过滤器后，可以多注意一下它们的用法。

另外一些时候，我们可能只是对表格的结构不满意，希望自己组织列；或者不希望有标题，这样方便其它程序解析结果等，这就用到了 Go 的模板语法。

比如，下面的命令会直接列出镜像结果，并且只包含镜像ID和仓库名：

```
$ docker images --format "{{.ID}}: {{.Repository}}"
5f515359c7f8: redis
05a60462f8ba: nginx
fe9198c04d62: mongo
00285df0df87: <none>
f753707788c5: ubuntu
f753707788c5: ubuntu
1e0c3dd64ccd: ubuntu
```

或者打算以表格等距显示，并且有标题行，和默认一样，不过自己定义列：

```
$ docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag
}}"
IMAGE ID            REPOSITORY          TAG
5f515359c7f8        redis               latest
05a60462f8ba        nginx               latest
fe9198c04d62        mongo               3.2
00285df0df87        <none>              <none>
f753707788c5        ubuntu              16.04
f753707788c5        ubuntu              latest
1e0c3dd64ccd        ubuntu              14.04
```

利用 commit 理解镜像构成

镜像是容器的基础，每次执行 `docker run` 的时候都会指定哪个镜像作为容器运行的基础。在之前的例子中，我们所使用的都是来自于 Docker Hub 的镜像。直接使用这些镜像是可以满足一定的需求，而当这些镜像无法直接满足需求时，我们就需要定制这些镜像。接下来的几节就将讲解如何定制镜像。

回顾一下之前我们学到的知识，镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，是在以镜像为基础层，在其基础上加一层作为容器运行时的存储层。

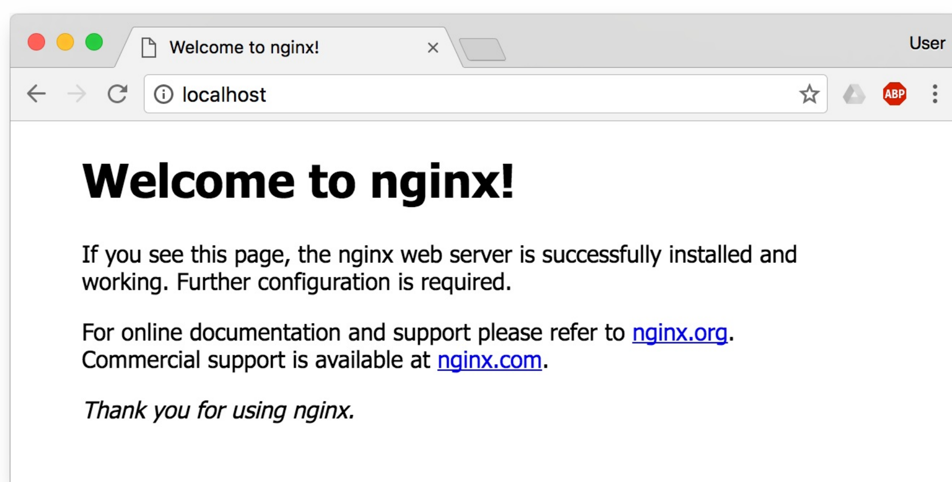
现在让我们以定制一个 Web 服务器为例子，来讲解镜像是如何构建的。

```
docker run --name webserver -d -p 80:80 nginx
```

这条命令会用 `nginx` 镜像启动一个容器，命名为 `webserver`，并且映射了 80 端口，这样我们可以用浏览器去访问这个 `nginx` 服务器。

如果是在 Linux 本机运行的 Docker，或者如果使用的是 Docker for Mac、Docker for Windows，那么可以直接访问：http://localhost；如果使用的是 Docker Toolbox，或者是在虚拟机、云服务器上安装的 Docker，则需要将 `localhost` 换为虚拟机地址或者实际云服务器地址。

直接用浏览器访问的话，我们会看到默认的 Nginx 欢迎页面。



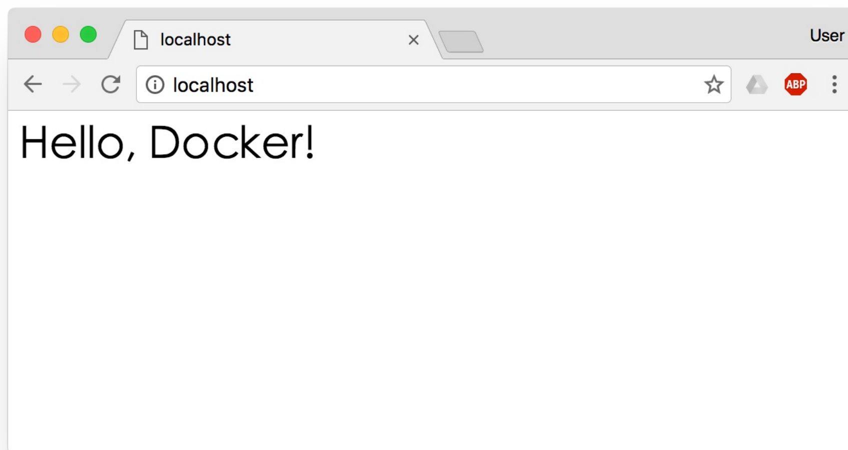
现在，假设我们非常不喜欢这个欢迎页面，我们希望改成欢迎 Docker 的文字，我们可以使用 `docker exec` 命令进入容器，修改其内容。

```
$ docker exec -it webserver bash
root@3729b97e8226:/# echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
root@3729b97e8226:/# exit
exit
```

我们以交互式终端方式进入 `webserver` 容器，并执行了 `bash` 命令，也就是获得一个可操作的 Shell。

然后，我们用 `<h1>Hello, Docker!</h1>` 覆盖了 `/usr/share/nginx/html/index.html` 的内容。

现在我们再刷新浏览器的话，会发现内容被改变了。



我们修改了容器的文件，也就是改动了容器的存储层。我们可以通过 `docker diff` 命令看到具体的改动。

```
$ docker diff webserver
C /root
A /root/.bash_history
C /run
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
C /var
C /var/cache
C /var/cache/nginx
A /var/cache/nginx/client_temp
A /var/cache/nginx/fastcgi_temp
A /var/cache/nginx/proxy_temp
A /var/cache/nginx/scgi_temp
A /var/cache/nginx/uwsgi_temp
```

现在我们定制好了变化，我们希望能将其保存下来形成镜像。

要知道，当我们运行一个容器的时候（如果不使用卷的话），我们做的任何文件修改都会被记录于容器存储层里。而 Docker 提供了一个 `docker commit` 命令，可以将容器的存储层保存下来成为镜像。换句话说，就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候，就会拥有原有容器最后的文件变化。

`docker commit` 的语法格式为：

```
docker commit [选项] <容器ID或容器名> [<仓库名>[:<标签>]]
```

我们可以用下面的命令将容器保存为镜像：

```
$ docker commit \
    --author "Tao Wang <twang2218@gmail.com>" \
    --message "修改了默认网页" \
    webserver \
    nginx:v2
sha256:07e33465974800ce65751acc279adc6ed2dc5ed4e0838f8b86f0c87aa
1795214
```

其中 `--author` 是指定修改的作者，而 `--message` 则是记录本次修改的内容。这点和 `git` 版本控制相似，不过这里这些信息可以省略留空。

我们可以在 `docker images` 中看到这个新定制的镜像：

```
$ docker images nginx
REPOSITORY          TAG                 IMAGE ID            CREA
TED             SIZE
nginx               v2                  07e334659748        9 se
conds ago       181.5 MB
nginx               1.11                05a60462f8ba        12 d
ays ago         181.5 MB
nginx               latest              e43d811ce2f4        4 we
eks ago         181.5 MB
```

我们还可以用 `docker history` 具体查看镜像内的历史记录，如果比较 `nginx:latest` 的历史记录，我们会发现新增了我们刚刚提交的这一层。

```
$ docker history nginx:v2
IMAGE               CREATED             CREATED BY
                        SIZE                COMMENT
07e334659748        54 seconds ago      nginx -g daemon off;
                        95 B                修改了默认网页
e43d811ce2f4        4 weeks ago         /bin/sh -c #(nop)  CMD [
"nginx" "-g" "daemon    0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  EXPOS
E 443/tcp 80/tcp        0 B
<missing>           4 weeks ago         /bin/sh -c ln -sf /dev/s
tdout /var/log/nginx/   22 B
<missing>           4 weeks ago         /bin/sh -c apt-key adv -
-keyserver hkp://pgp.   58.46 MB
<missing>           4 weeks ago         /bin/sh -c #(nop)  ENV N
GINX_VERSION=1.11.5-1   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  MAINT
AINER NGINX Docker Ma   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  CMD [
"/bin/bash"]            0 B
<missing>           4 weeks ago         /bin/sh -c #(nop) ADD fi
le:23aa4f893e3288698c   123 MB
```

新的镜像定制好后，我们可以来运行这个镜像。

```
docker run --name web2 -d -p 81:80 nginx:v2
```

这里我们命名为新的服务为 `web2` ，并且映射到 `81` 端口。如果是 Docker for Mac/Windows 或 Linux 桌面的话，我们就可以直接访问 http://localhost:81 看到结果，其内容应该和之前修改后的 `webserver` 一样。

至此，我们第一次完成了定制镜像，使用的是 `docker commit` 命令，手动操作给旧的镜像添加了新的一层，形成新的镜像，对镜像多层存储应该有了更直观的感觉。

慎用 `docker commit`

使用 `docker commit` 命令虽然可以比较直观的帮助理解镜像分层存储的概念，但是实际环境中并不会这样使用。

首先，如果仔细观察之前的 `docker diff webserver` 的结果，你会发现除了真正想要修改的 `/usr/share/nginx/html/index.html` 文件外，由于命令的执行，还有很多文件被改动或添加了。这还仅仅是最简单的操作，如果是安装软件包、编译构建，那会有大量的无关内容被添加进来，如果不小心清理，将会导致镜像极为臃肿。

此外，使用 `docker commit` 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被称为黑箱镜像，换句话说，就是除了制作镜像的人知道执行过什么命令、怎么生成的镜像，别人根本无从得知。而且，即使是这个制作镜像的人，过一段时间后也无法记清具体在操作的。虽然 `docker diff` 或许可以告诉得到一些线索，但是远远不到可以确保生成一致镜像的地步。这种黑箱镜像的维护工作是非常痛苦的。

而且，回顾之前提及的镜像所使用的分层存储的概念，除当前层外，之前的每一层都是不会发生改变的，换句话说，任何修改的结果仅仅是在当前层进行标记、添加、修改，而不会改动上一层。如果使用 `docker commit` 制作镜像，以及后期修改的话，每一次修改都会让镜像更加臃肿一次，所删除的上一层的东西并不会丢失，会一直如影随形的跟着这个镜像，即使根本无法访问到™。这会让镜像更加臃肿。

`docker commit` 命令除了学习之外，还有一些特殊的应用场合，比如被入侵后保存现场等。但是，不要使用 `docker commit` 定制镜像，定制行为应该使用 `Dockerfile` 来完成。下面的章节我们就来讲述一下如何使用 `Dockerfile` 定制镜像。

使用 Dockerfile 定制镜像

从刚才的 `docker commit` 的学习中，我们可以了解到，镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 Dockerfile。

Dockerfile 是一个文本文件，其内包含了一条条的指令**(Instruction)**，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

还以之前定制 `nginx` 镜像为例，这次我们使用 Dockerfile 来定制。

在一个空白目录中，建立一个文本文件，并命名为 `Dockerfile` ：

```
$ mkdir mynginx
$ cd mynginx
$ touch Dockerfile
```

其内容为：

```
FROM nginx
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index
.html
```

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令， `FROM` 和 `RUN` 。

FROM 指定基础镜像

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 `nginx` 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 `FROM` 就是指定基础镜像，因此一个 `Dockerfile` 中 `FROM` 是必备的指令，并且必须是第一条指令。

在 Docker Hub[1] 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如 `nginx`、`redis`、`mongo`、`mysql`、`httpd`、`php`、`tomcat` 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 `node`、`openjdk`、`python`、`ruby`、`golang` 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 `ubuntu`、`debian`、`centos`、`fedora`、`alpine` 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 `scratch`。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。

```
FROM scratch
...
```

如果你以 `scratch` 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。

不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，比如 `swarm`、`coreos/etcd`。对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 `FROM scratch` 会让镜像体积更加小巧。使用 Go 语言 开发的应用很多会使用这种方式来制作镜像，这也是为什么有人认为 Go 是特别适合容器微服务架构的语言的原因之一。

RUN 执行命令

`RUN` 指令是用来执行命令行命令的。由于命令行的强大能力，`RUN` 指令在定制镜像时是最常用的指令之一。其格式有两种：

- *shell* 格式：`RUN <命令>`，就像直接在命令行中输入的命令一样。刚才写的 Dockerfile 中的 `RUN` 指令就是这种格式。

```
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index
.html
```

- *exec* 格式：`RUN ["可执行文件", "参数1", "参数2"]`，这更像是函数调用中的格式。

既然 `RUN` 就像 Shell 脚本一样可以执行命令，那么我们是否就可以像 Shell 脚本一样把每个命令对应一个 RUN 呢？比如这样：

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y gcc libc6-dev make
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz"
RUN mkdir -p /usr/src/redis
RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1
RUN make -C /usr/src/redis
RUN make -C /usr/src/redis install
```

之前说过，Dockerfile 中每一个指令都会建立一层，`RUN` 也不例外。每一个 `RUN` 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，`commit` 这一层的修改，构成新的镜像。

而上面的这种写法，创建了 7 层镜像。这是完全没有意义的，而且很多运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常臃肿、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错。 这是很多初学 Docker 的人常犯的一个错误。

Union FS 是有最大层数限制的，比如 *AUFS*，曾经是最大不得超过 *42* 层，现在是不得超过 *127* 层。

上面的 `Dockerfile` 正确的写法应该是这样：

```
FROM debian:jessie

RUN buildDeps='gcc libc6-dev make' \
    && apt-get update \
    && apt-get install -y $buildDeps \
    && wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz" \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -rf /var/lib/apt/lists/* \
    && rm redis.tar.gz \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps
```

首先，之前所有的命令只有一个目的，就是编译、安装 redis 可执行文件。因此没有必要建立很多层，这只是一层的事情。因此，这里没有使用很多个 `RUN` 对一一对应不同的命令，而是仅仅使用一个 `RUN` 指令，并使用 `&&` 将各个所需命令串联起来。将之前的 7 层，简化为了 1 层。在撰写 Dockerfile 的时候，要经常提醒自己，这并不是在写 Shell 脚本，而是在定义每一层该如何构建。

并且，这里为了格式化还进行了换行。Dockerfile 支持 Shell 类的行尾添加 `\` 的命令换行方式，以及行首 `#` 进行注释的格式。良好的格式，比如换行、缩进、注释等，会让维护、排障更为容易，这是一个比较好的习惯。

此外，还可以看到这一组命令的最后添加了清理工作的命令，删除了为了编译构建所需要的软件，清理了所有下载、展开的文件，并且还清理了 `apt` 缓存文件。这是很重要的一步，我们之前说过，镜像是多层存储，每一层的东西并不会在下一层被删除，会一直跟随着镜像。因此镜像构建时，一定要确保每一层只添加真正需要添加的东西，任何无关的东西都应该清理掉。

很多人初学 Docker 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

构建镜像

好了，让我们再回到之前定制的 nginx 镜像的 Dockerfile 来。现在我们明白了这个 Dockerfile 的内容，那么让我们来构建这个镜像吧。

在 `Dockerfile` 文件所在目录执行：

```
$ docker build -t nginx:v3 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM nginx
 ---> e43d811ce2f4
Step 2 : RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/h
tml/index.html
 ---> Running in 9cdc27646c7b
 ---> 44aa4490ce2c
Removing intermediate container 9cdc27646c7b
Successfully built 44aa4490ce2c
```

从命令的输出结果中，我们可以清晰的看到镜像的构建过程。在 `Step 2` 中，如同我们之前所说的那样， `RUN` 指令启动了一个容器 `9cdc27646c7b` ，执行了所要求的命令，并最后提交了这一层 `44aa4490ce2c` ，随后删除了所用到的这个容器 `9cdc27646c7b` 。

这里我们使用了 `docker build` 命令进行镜像构建。其格式为：

```
docker build [选项] <上下文路径/URL/->
```

在这里我们指定了最终镜像的名称 `-t nginx:v3` ，构建成功后，我们可以像之前运行 `nginx:v2` 那样来运行这个镜像，其结果会和 `nginx:v2` 一样。

镜像构建上下文（Context）

如果注意，会看到 `docker build` 命令最后有一个 `.` 。 `.` 表示当前目录，而 `Dockerfile` 就在当前目录，因此不少初学者以为这个路径是在指定 `Dockerfile` 所在路径，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定上下文路径。那么什么是上下文呢？

首先我们要理解 `docker build` 的工作原理。Docker 在运行时分为 Docker 引擎（也就是服务端守护进程）和客户端工具。Docker 的引擎提供了一组 REST API，被称为 Docker Remote API，而如 `docker` 命令这样的客户端工具，则是通过这

组 API 与 Docker 引擎交互，从而完成各种功能。因此，虽然表面上我们好像是在本机执行各种 `docker` 功能，但实际上，一切都是使用的远程调用形式在服务端（Docker 引擎）完成。也因为这种 C/S 设计，让我们操作远程服务器的 Docker 引擎变得轻而易举。

当我们进行镜像构建的时候，并非所有定制都会通过 `RUN` 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 `COPY` 指令、`ADD` 指令等。而 `docker build` 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？

这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，`docker build` 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。

如果在 `Dockerfile` 中这么写：

```
COPY ./package.json /app/
```

这并不是要复制执行 `docker build` 命令所在的目录下的 `package.json`，也不是复制 `Dockerfile` 所在目录下的 `package.json`，而是复制 上下文（**context**） 目录下的 `package.json`。

因此，`COPY` 这类指令中的源文件的路径都是相对路径。这也是初学者经常会问的为什么 `COPY ../package.json /app` 或者 `COPY /opt/xxxx /app` 无法工作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位置的文件。如果真的需要那些文件，应该将它们复制到上下文目录中去。

现在就可以理解刚才的命令 `docker build -t nginx:v3 .` 中的这个 `.`，实际上是在指定上下文的目录，`docker build` 命令会将该目录下的内容打包交给 Docker 引擎以帮助构建镜像。

如果观察 `docker build` 输出，我们其实已经看到了这个发送上下文的过程：

```
$ docker build -t nginx:v3 .
Sending build context to Docker daemon 2.048 kB
...
```

理解构建上下文对于镜像构建是很重要的，避免犯一些不应该的错误。比如有些初学者在发现 `COPY /opt/xxxx /app` 不工作后，于是干脆将 `Dockerfile` 放到了硬盘根目录去构建，结果发现 `docker build` 执行后，在发送一个几十 GB 的东西，极为缓慢而且很容易构建失败。那是因为这种做法是在让 `docker build` 打包整个硬盘，这显然是使用错误。

一般来说，应该会将 `Dockerfile` 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 `.gitignore` 一样的语法写一个 `.dockerignore`，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

那么为什么会有人误以为 `.` 是指定 `Dockerfile` 所在目录呢？这是因为在默认情况下，如果不额外指定 `Dockerfile` 的话，会将上下文目录下的名为 `Dockerfile` 的文件作为 Dockerfile。

这只是默认行为，实际上 `Dockerfile` 的文件名并不要求必须为 `Dockerfile`，而且并不要求必须位于上下文目录中，比如可以用 `-f ../Dockerfile.php` 参数指定某个文件作为 `Dockerfile`。

当然，一般大家习惯性的会使用默认的文件名 `Dockerfile`，以及会将其置于镜像构建上下文目录中。

其它 `docker build` 的用法

直接用 Git repo 进行构建

或许你已经注意到了，`docker build` 还支持从 URL 构建，比如可以直接从 Git repo 中构建：

```
$ docker build https://github.com/twang2218/gitlab-ce-zh.git#:8.
14
docker build https://github.com/twang2218/gitlab-ce-zh.git\#:8.14

Sending build context to Docker daemon 2.048 kB
Step 1 : FROM gitlab/gitlab-ce:8.14.0-ce.0
8.14.0-ce.0: Pulling from gitlab/gitlab-ce
aed15891ba52: Already exists
773ae8583d14: Already exists
...
```

这行命令指定了构建所需的 Git repo，并且指定默认的 `master` 分支，构建目录为 `/8.14/` ，然后 Docker 就会自己去 `git clone` 这个项目、切换到指定分支、并进入到指定目录后开始构建。

用给定的 tar 压缩包构建

```
$ docker build http://server/context.tar.gz
```

如果所给出的 URL 不是个 Git repo，而是个 `tar` 压缩包，那么 Docker 引擎会下载这个包，并自动解压缩，以其作为上下文，开始构建。

从标准输入中读取 Dockerfile 进行构建

```
docker build - < Dockerfile
```

或

```
cat Dockerfile | docker build -
```

如果标准输入传入的是文本文件，则将其视为 `Dockerfile` ，并开始构建。这种形式由于直接从标准输入中读取 Dockerfile 的内容，它没有上下文，因此不可以像其他方法那样可以将本地文件 `COPY` 进镜像之类的事情。

从标准输入中读取上下文压缩包进行构建

```
$ docker build - < context.tar.gz
```

如果发现标准输入的文件格式是 `gzip` 、 `bzip2` 以及 `xz` 的话，将会使其为上下文压缩包，直接将其展开，将里面视为上下文，并开始构建。

> 1. Docker Store是发现公共Docker内容，镜像发布和发行软件的新地方 ↩

Dockerfile 指令详解

我们已经介绍了 `FROM`，`RUN`，还提及了 `COPY`, `ADD`，其实 Dockerfile 功能很强大，它提供了十多个指令。这里我们继续讲解剩下的指令。

COPY 复制文件

格式：

- COPY <源路径>... <目标路径>
- COPY ["<源路径1>",... "<目标路径>"]

和 RUN 指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。

COPY 指令将从构建上下文目录中 <源路径> 的文件/目录复制到新的一层的镜像内的 <目标路径> 位置。比如：

```
COPY package.json /usr/src/app/
```

<源路径> 可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的 filepath.Match 规则，如：

```
COPY hom* /mydir/
COPY hom?.txt /mydir/
```

<目标路径> 可以是容器内的绝对路径，也可以是相对于工作目录的相对路径（工作目录可以用 WORKDIR 指令来指定）。目标路径不需要事先创建，如果目录不存在会在复制文件前先行创建缺失目录。

此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限、文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git 进行管理的时候。

ADD 更高级的复制文件

`ADD` 指令和 `COPY` 的格式和性质基本一致。但是在 `COPY` 基础上增加了一些功能。

比如 `<源路径>` 可以是一个 `URL`，这种情况下，Docker 引擎会试图去下载这个链接的文件放到 `<目标路径>` 去。下载后的文件权限自动设置为 `600`，如果这并不是想要的权限，那么还需要增加额外的一层 `RUN` 进行权限调整，另外，如果下载的是个压缩包，需要解压缩，也一样还需要额外的一层 `RUN` 指令进行解压缩。所以不如直接使用 `RUN` 指令，然后使用 `wget` 或者 `curl` 工具下载，处理权限、解压缩、然后清理无用文件更合理。因此，这个功能其实并不实用，而且不推荐使用。

如果 `<源路径>` 为一个 `tar` 压缩文件的话，压缩格式为 `gzip`, `bzip2` 以及 `xz` 的情况下，`ADD` 指令将会自动解压缩这个压缩文件到 `<目标路径>` 去。

在某些情况下，这个自动解压缩的功能非常有用，比如官方镜像 `ubuntu` 中：

```
FROM scratch
ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /
...
```

但在某些情况下，如果我们真的是希望复制个压缩文件进去，而不解压缩，这时就不可以使用 `ADD` 命令了。

在 Docker 官方的最佳实践文档中要求，尽可能的使用 `COPY`，因为 `COPY` 的语义很明确，就是复制文件而已，而 `ADD` 则包含了更复杂的功能，其行为也不一定很清晰。最适合使用 `ADD` 的场合，就是所提及的需要自动解压缩的场合。

另外需要注意的是，`ADD` 指令会令镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢。

因此在 `COPY` 和 `ADD` 指令中选择的时候，可以遵循这样的原则，所有的文件复制均使用 `COPY` 指令，仅在需要自动解压缩的场合使用 `ADD`。

CMD 容器启动命令

`CMD` 指令的格式和 `RUN` 相似，也是两种格式：

- `shell` 格式：`CMD <命令>`
- `exec` 格式：`CMD ["可执行文件", "参数1", "参数2"...]`
- 参数列表格式：`CMD ["参数1", "参数2"...]`。在指定了 `ENTRYPOINT` 指令后，用 `CMD` 指定具体的参数。

之前介绍容器的时候曾经说过，Docker 不是虚拟机，容器就是进程。既然是进程，那么在启动容器的时候，需要指定所运行的程序及参数。`CMD` 指令就是用于指定默认的容器主进程的启动命令的。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令，比如，`ubuntu` 镜像默认的 `CMD` 是 `/bin/bash`，如果我们直接 `docker run -it ubuntu` 的话，会直接进入 `bash`。我们也可以在运行时指定运行别的命令，如 `docker run -it ubuntu cat /etc/os-release`。这就是用 `cat /etc/os-release` 命令替换了默认的 `/bin/bash` 命令了，输出了系统版本信息。

在指令格式上，一般推荐使用 `exec` 格式，这类格式在解析时会被解析为 JSON 数组，因此一定要使用双引号 `"`，而不要使用单引号。

如果使用 `shell` 格式的话，实际的命令会被包装为 `sh -c` 的参数的形式进行执行。比如：

```
CMD echo $HOME
```

在实际执行中，会将其变更为：

```
CMD [ "sh", "-c", "echo $HOME" ]
```

这就是为什么我们可以使用环境变量的原因，因为这些环境变量会被 shell 进行解析处理。

提到 `CMD` 就不得不提容器中应用在前台执行和后台执行的问题。这是初学者常出现的一个混淆。

Docker 不是虚拟机，容器中的应用都应该以前台执行，而不是像虚拟机、物理机里面那样，用 upstart/systemd 去启动后台服务，容器内没有后台服务的概念。

一些初学者将 `CMD` 写为：

```
CMD service nginx start
```

然后发现容器执行后就立即退出了。甚至在容器内去使用 `systemctl` 命令结果却发现根本执行不了。这就是因为没有搞明白前台、后台的概念，没有区分容器和虚拟机的差异，依旧在以传统虚拟机的角度去理解容器。

对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其它辅助进程不是它需要关心的东西。

而使用 `service nginx start` 命令，则是希望 upstart 来以后台守护进程形式启动 `nginx` 服务。而刚才说了 `CMD service nginx start` 会被理解为 `CMD [ "sh", "-c", "service nginx start"]` ，因此主进程实际上是 `sh` 。那么当 `service nginx start` 命令结束后， `sh` 也就结束了， `sh` 作为主进程退出了，自然就会令容器退出。

正确的做法是直接执行 `nginx` 可执行文件，并且要求以前台形式运行。比如：

```
CMD ["nginx", "-g", "daemon off;"]
```

ENTRYPOINT 入口点

`ENTRYPOINT` 的格式和 `RUN` 指令格式一样，分为 `exec` 格式和 `shell` 格式。

`ENTRYPOINT` 的目的和 `CMD` 一样，都是在指定容器启动程序及参数。`ENTRYPOINT` 在运行时也可以替代，不过比 `CMD` 要略显繁琐，需要通过 `docker run` 的参数 `--entrypoint` 来指定。

当指定了 `ENTRYPOINT` 后，`CMD` 的含义就发生了改变，不再是直接的运行其命令，而是将 `CMD` 的内容作为参数传给 `ENTRYPOINT` 指令，换句话说实际执行时，将变为：

```
<ENTRYPOINT> "<CMD>"
```

那么有了 `CMD` 后，为什么还要有 `ENTRYPOINT` 呢？这种 `<ENTRYPOINT> "<CMD>"` 有什么好处么？让我们来看几个场景。

场景一：让镜像变成像命令一样使用

假设我们需要一个得知自己当前公网 IP 的镜像，那么可以先用 `CMD` 来实现：

```
FROM ubuntu:16.04
RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*
CMD [ "curl", "-s", "http://ip.cn" ]
```

假如我们使用 `docker build -t myip .` 来构建镜像的话，如果我们需要查询当前公网 IP，只需要执行：

```
$ docker run myip
当前 IP：61.148.226.66 来自：北京市 联通
```

嗯，这么看起来好像可以直接把镜像当做命令使用了，不过命令总有参数，如果我们希望加参数呢？比如从上面的 `CMD` 中可以看到实质的命令是 `curl`，那么如果我们希望显示 HTTP 头信息，就需要加上 `-i` 参数。那么我们可以直接加 `-i` 参数给 `docker run myip` 么？

```
$ docker run myip -i
docker: Error response from daemon: invalid header field value "
oci runtime error: container_linux.go:247: starting container pr
ocess caused \"exec: \\\"-i\\\": executable file not found in $P
ATH\"\n".
```

我们可以看到可执行文件找不到的报错，`executable file not found`。之前我们说过，跟在镜像名后面的是 `command`，运行时会替换 `CMD` 的默认值。因此这里的 `-i` 替换了原来的 `CMD`，而不是添加在原来的 `curl -s http://ip.cn` 后面。而 `-i` 根本不是命令，所以自然找不到。

那么如果我们希望加入 `-i` 这参数，我们就必须重新完整的输入这个命令：

```
$ docker run myip curl -s http://ip.cn -i
```

这显然不是很好的解决方案，而使用 `ENTRYPOINT` 就可以解决这个问题。现在我们重新用 `ENTRYPOINT` 来实现这个镜像：

```
FROM ubuntu:16.04
RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*
ENTRYPOINT [ "curl", "-s", "http://ip.cn" ]
```

这次我们再来尝试直接使用 `docker run myip -i`：

```
$ docker run myip
当前 IP：61.148.226.66 来自：北京市 联通

$ docker run myip -i
HTTP/1.1 200 OK
Server: nginx/1.8.0
Date: Tue, 22 Nov 2016 05:12:40 GMT
Content-Type: text/html; charset=UTF-8
Vary: Accept-Encoding
X-Powered-By: PHP/5.6.24-1~dotdeb+7.1
X-Cache: MISS from cache-2
X-Cache-Lookup: MISS from cache-2:80
X-Cache: MISS from proxy-2_6
Transfer-Encoding: chunked
Via: 1.1 cache-2:80, 1.1 proxy-2_6:8006
Connection: keep-alive

当前 IP：61.148.226.66 来自：北京市 联通
```

可以看到，这次成功了。这是因为当存在 `ENTRYPOINT` 后， `CMD` 的内容将会作为参数传给 `ENTRYPOINT` ，而这里 `-i` 就是新的 `CMD` ，因此会作为参数传给 `curl` ，从而达到了我们预期的效果。

场景二：应用运行前的准备工作

启动容器就是启动主进程，但有些时候，启动主进程前，需要一些准备工作。

比如 `mysql` 类的数据库，可能需要一些数据库配置、初始化的工作，这些工作要在最终的 mysql 服务器运行之前解决。

此外，可能希望避免使用 `root` 用户去启动服务，从而提高安全性，而在启动服务前还需要以 `root` 身份执行一些必要的准备工作，最后切换到服务用户身份启动服务。或者除了服务外，其它命令依旧可以使用 `root` 身份执行，方便调试等。

这些准备工作是和容器 `CMD` 无关的，无论 `CMD` 为什么，都需要事先进行一个预处理的工作。这种情况下，可以写一个脚本，然后放入 `ENTRYPOINT` 中去执行，而这个脚本会将接到的参数（也就是 `<CMD>` ）作为命令，在脚本最后执行。比如官方镜像 `redis` 中就是这么做的：

```
FROM alpine:3.4
...
RUN addgroup -S redis && adduser -S -G redis redis
...
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 6379
CMD [ "redis-server" ]
```

可以看到其中为了 redis 服务创建了 redis 用户，并在最后指定了 `ENTRYPOINT` 为 `docker-entrypoint.sh` 脚本。

```
#!/bin/sh
...
# allow the container to be started with `--user`
if [ "$1" = 'redis-server' -a "$(id -u)" = '0' ]; then
    chown -R redis .
    exec su-exec redis "$0" "$@"
fi

exec "$@"
```

该脚本的内容就是根据 `CMD` 的内容来判断，如果是 `redis-server` 的话，则切换到 `redis` 用户身份启动服务器，否则依旧使用 `root` 身份执行。比如：

```
$ docker run -it redis id
uid=0(root) gid=0(root) groups=0(root)
```

ENV 设置环境变量

格式有两种：

- ENV <key> <value>
- ENV <key1>=<value1> <key2>=<value2>...

这个指令很简单，就是设置环境变量而已，无论是后面的其它指令，如 `RUN` ，还是运行时的应用，都可以直接使用这里定义的环境变量。

```
ENV VERSION=1.0 DEBUG=on \
    NAME="Happy Feet"
```

这个例子中演示了如何换行，以及对含有空格的值用双引号括起来的办法，这和 Shell 下的行为是一致的。

定义了环境变量，那么在后续的指令中，就可以使用这个环境变量。比如在官方 `node` 镜像 `Dockerfile` 中，就有类似这样的代码：

```
ENV NODE_VERSION 7.2.0

RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-linux-x64.tar.xz" \
  && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
  && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
  && grep " node-v$NODE_VERSION-linux-x64.tar.xz\$" SHASUMS256.txt | sha256sum -c - \
  && tar -xJf "node-v$NODE_VERSION-linux-x64.tar.xz" -C /usr/local --strip-components=1 \
  && rm "node-v$NODE_VERSION-linux-x64.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt \
  && ln -s /usr/local/bin/node /usr/local/bin/nodejs
```

在这里先定义了环境变量 `NODE_VERSION` ，其后的 `RUN` 这层里，多次使用 `$NODE_VERSION` 来进行操作定制。可以看到，将来升级镜像构建版本的时候，只需要更新 `7.2.0` 即可， `Dockerfile` 构建维护变得更轻松了。

下列指令可以支持环境变量展开：

`ADD` 、 `COPY` 、 `ENV` 、 `EXPOSE` 、 `LABEL` 、 `USER` 、 `WORKDIR` 、 `VOLUME` 、 `STOPSIGNAL` 、 `ONBUILD` 。

可以从这个指令列表里感觉到，环境变量可以使用的地方很多，很强大。通过环境变量，我们可以让一份 `Dockerfile` 制作更多的镜像，只需使用不同的环境变量即可。

ARG 构建参数

格式： `ARG <参数名>[=<默认值>]`

构建参数和 `ENV` 的效果一样，都是设置环境变量。所不同的是， `ARG` 所设置的构建环境的环境变量，在将来容器运行时是不会存在这些环境变量的。但是不要因此就使用 `ARG` 保存密码之类的信息，因为 `docker history` 还是可以看到所有值的。

`Dockerfile` 中的 `ARG` 指令是定义参数名称，以及定义其默认值。该默认值可以在构建命令 `docker build` 中用 `--build-arg <参数名>=<值>` 来覆盖。

在 1.13 之前的版本，要求 `--build-arg` 中的参数名，必须在 `Dockerfile` 中用 `ARG` 定义过了，换句话说，就是 `--build-arg` 指定的参数，必须在 `Dockerfile` 中使用了。如果对应参数没有被使用，则会报错退出构建。从 1.13 开始，这种严格的限制被放开，不再报错退出，而是显示警告信息，并继续构建。这对于使用 CI 系统，用同样的构建流程构建不同的 `Dockerfile` 的时候比较有帮助，避免构建命令必须根据每个 Dockerfile 的内容修改。

VOLUME 定义匿名卷

格式为：

- `VOLUME ["<路径1>", "<路径2>"...]`
- `VOLUME <路径>`

之前我们说过，容器运行时应该尽量保持容器存储层不发生写操作，对于数据库类需要保存动态数据的应用，其数据库文件应该保存于卷(volume)中，后面的章节我们会进一步介绍 Docker 卷的概念。为了防止运行时用户忘记将动态文件所保存目录挂载为卷，在 `Dockerfile` 中，我们可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

```
VOLUME /data
```

这里的 `/data` 目录就会在运行时自动挂载为匿名卷，任何向 `/data` 中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。当然，运行时可以覆盖这个挂载设置。比如：

```
docker run -d -v mydata:/data xxxx
```

在这行命令中，就使用了 `mydata` 这个命名卷挂载到了 `/data` 这个位置，替代了 `Dockerfile` 中定义的匿名卷的挂载配置。

EXPOSE 声明端口

格式为 `EXPOSE <端口1> [<端口2>...]` 。

`EXPOSE` 指令是声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是 `docker run -P` 时，会自动随机映射 `EXPOSE` 的端口。

此外，在早期 Docker 版本中还有一个特殊的用处。以前所有容器都运行于默认桥接网络中，因此所有容器互相之间都可以直接访问，这样存在一定的安全性问题。于是有了一个 Docker 引擎参数 `--icc=false` ，当指定该参数后，容器间将默认无法互访，除非互相间使用了 `--links` 参数的容器才可以互通，并且只有镜像中 `EXPOSE` 所声明的端口才可以被访问。这个 `--icc=false` 的用法，在引入了 `docker network` 后已经基本不用了，通过自定义网络可以很轻松的实现容器间的互联与隔离。

要将 `EXPOSE` 和在运行时使用 `-p <宿主端口>:<容器端口>` 区分开来。 `-p` ，是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而 `EXPOSE` 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

WORKDIR 指定工作目录

格式为 `WORKDIR <工作目录路径>` 。

使用 `WORKDIR` 指令可以来指定工作目录（或者称为当前目录），以后各层的当前目录就被改为指定的目录，如该目录不存在， `WORKDIR` 会帮你建立目录。

之前提到一些初学者常犯的错误是把 `Dockerfile` 等同于 Shell 脚本来书写，这种错误的理解还可能会导致出现下面这样的错误：

```
RUN cd /app
RUN echo "hello" > world.txt
```

如果将这个 Dockerfile 进行构建镜像运行后，会发现找不到 `/app/world.txt` 文件，或者其内容不是 `hello` 。原因其实很简单，在 Shell 中，连续两行是同一个进程执行环境，因此前一个命令修改的内存状态，会直接影响后一个命令；而在 Dockerfile 中，这两行 `RUN` 命令的执行环境根本不同，是两个完全不同的容器。这就是对 Dokerfile 构建分层存储的概念不了解所导致的错误。

之前说过每一个 `RUN` 都是启动一个容器、执行命令、然后提交存储层文件变更。第一层 `RUN cd /app` 的执行仅仅是当前进程的工作目录变更，一个内存上的变化而已，其结果不会造成任何文件变更。而到第二层的时候，启动的是一个全新的容器，跟第一层的容器更完全没关系，自然不可能继承前一层构建过程中的内存变化。

因此如果需要改变以后各层的工作目录的位置，那么应该使用 `WORKDIR` 指令。

USER 指定当前用户

格式： `USER <用户名>`

`USER` 指令和 `WORKDIR` 相似，都是改变环境状态并影响以后的层。`WORKDIR` 是改变工作目录，`USER` 则是改变之后层的执行 `RUN`, `CMD` 以及 `ENTRYPOINT` 这类命令的身份。

当然，和 `WORKDIR` 一样，`USER` 只是帮助你切换到指定用户而已，这个用户必须是事先建立好的，否则无法切换。

```
RUN groupadd -r redis && useradd -r -g redis redis
USER redis
RUN [ "redis-server" ]
```

如果以 `root` 执行的脚本，在执行期间希望改变身份，比如希望以某个已经建立好的用户来运行某个服务进程，不要使用 `su` 或者 `sudo`，这些都需要比较麻烦的配置，而且在 TTY 缺失的环境下经常出错。建议使用 `gosu`，可以从其项目网站看到进一步的信息：https://github.com/tianon/gosu

```
# 建立 redis 用户，并使用 gosu 换另一个用户执行命令
RUN groupadd -r redis && useradd -r -g redis redis
# 下载 gosu
RUN wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/
releases/download/1.7/gosu-amd64" \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true
# 设置 CMD，并以另外的用户执行
CMD [ "exec", "gosu", "redis", "redis-server" ]
```

HEALTHCHECK 健康检查

格式：

- `HEALTHCHECK [选项] CMD <命令>` ：设置检查容器健康状况的命令
- `HEALTHCHECK NONE` ：如果基础镜像有健康检查指令，使用这行可以屏蔽掉其健康检查指令

`HEALTHCHECK` 指令是告诉 Docker 应该如何进行判断容器的状态是否正常，这是 Docker 1.12 引入的新指令。

在没有 `HEALTHCHECK` 指令前，Docker 引擎只可以通过容器内主进程是否退出来判断容器是否状态异常。很多情况下这没问题，但是如果程序进入死锁状态，或者死循环状态，应用进程并不退出，但是该容器已经无法提供服务了。在 1.12 以前，Docker 不会检测到容器的这种状态，从而不会重新调度，导致可能会有部分容器已经无法提供服务了却还在接受用户请求。

而自 1.12 之后，Docker 提供了 `HEALTHCHECK` 指令，通过该指令指定一行命令，用这行命令来判断容器主进程的服务状态是否还正常，从而比较真实的反应容器实际状态。

当在一个镜像指定了 `HEALTHCHECK` 指令后，用其启动容器，初始状态会为 `starting` ，在 `HEALTHCHECK` 指令检查成功后变为 `healthy` ，如果连续一定次数失败，则会变为 `unhealthy` 。

`HEALTHCHECK` 支持下列选项：

- `--interval=<间隔>` ：两次健康检查的间隔，默认为 30 秒；
- `--timeout=<时长>` ：健康检查命令运行超时时间，如果超过这个时间，本次健康检查就被视为失败，默认 30 秒；
- `--retries=<次数>` ：当连续失败指定次数后，则将容器状态视为 `unhealthy` ，默认 3 次。

和 `CMD` , `ENTRYPOINT` 一样， `HEALTHCHECK` 只可以出现一次，如果写了多个，只有最后一个生效。

在 `HEALTHCHECK [选项] CMD` 后面的命令，格式和 `ENTRYPOINT` 一样，分为 `shell` 格式，和 `exec` 格式。命令的返回值决定了该次健康检查的成功与否： `0` ：成功； `1` ：失败； `2` ：保留，不要使用这个值。

假设我们有个镜像是个最简单的 Web 服务，我们希望增加健康检查来判断其 Web 服务是否在正常工作，我们可以用 `curl` 来帮助判断，其 `Dockerfile` 的 `HEALTHCHECK` 可以这么写：

```
FROM nginx
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib
/apt/lists/*
HEALTHCHECK --interval=5s --timeout=3s \
  CMD curl -fs http://localhost/ || exit 1
```

这里我们设置了每 5 秒检查一次（这里为了试验所以间隔非常短，实际应该相对较长），如果健康检查命令超过 3 秒没响应就视为失败，并且使用 `curl -fs http://localhost/ || exit 1` 作为健康检查命令。

使用 `docker build` 来构建这个镜像：

```
$ docker build -t myweb:v1 .
```

构建好了后，我们启动一个容器：

```
$ docker run -d --name web -p 80:80 myweb:v1
```

当运行该镜像后，可以通过 `docker ps` 看到最初的状态为 `(health: starting)`：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS                            PORTS
          NAMES
03e28eb00bd0        myweb:v1            "nginx -g 'daemon off"
 3 seconds ago       Up 2 seconds (health: starting)   80/tcp, 4
43/tcp     web
```

在等待几秒钟后，再次 `docker ps`，就会看到健康状态变化为了 `(healthy)`：

```
$ docker ps
CONTAINER ID          IMAGE                 COMMAND
 CREATED               STATUS                       PORTS
   NAMES
03e28eb00bd0          myweb:v1              "nginx -g 'daemon off"
 18 seconds ago        Up 16 seconds (healthy)   80/tcp, 443/tcp
   web
```

如果健康检查连续失败超过了重试次数，状态就会变为 `(unhealthy)` 。

为了帮助排障，健康检查命令的输出（包括 `stdout` 以及 `stderr` ）都会被存储于健康状态里，可以用 `docker inspect` 来查看。

```
$ docker inspect --format '{{json .State.Health}}' web | python
-m json.tool
{
    "FailingStreak": 0,
    "Log": [
        {
            "End": "2016-11-25T14:35:37.940957051Z",
            "ExitCode": 0,
            "Output": "<!DOCTYPE html>\n<html>\n<head>\n<title>W
elcome to nginx!</title>\n<style>\n    body {\n        width: 35
em;\n        margin: 0 auto;\n        font-family: Tahoma, Verda
na, Arial, sans-serif;\n    }\n</style>\n</head>\n<body>\n<h1>We
lcome to nginx!</h1>\n<p>If you see this page, the nginx web ser
ver is successfully installed and\nworking. Further configuratio
n is required.</p>\n\n<p>For online documentation and support pl
ease refer to\n<a href=\"http://nginx.org/\">nginx.org</a>.<br/>
\nCommercial support is available at\n<a href=\"http://nginx.com
/\">nginx.com</a>.</p>\n\n<p><em>Thank you for using nginx.</em>
</p>\n</body>\n</html>\n",
            "Start": "2016-11-25T14:35:37.780192565Z"
        }
    ],
    "Status": "healthy"
}
```

ONBUILD 为他人做嫁衣裳

格式：`ONBUILD <其它指令>`。

`ONBUILD` 是一个特殊的指令，它后面跟的是其它指令，比如 `RUN`, `COPY` 等，而这些指令，在当前镜像构建时并不会被执行。只有当以当前镜像为基础镜像，去构建下一级镜像的时候才会被执行。

`Dockerfile` 中的其它指令都是为了定制当前镜像而准备的，唯有 `ONBUILD` 是为了帮助别人定制自己而准备的。

假设我们要制作 Node.js 所写的应用的镜像。我们都知道 Node.js 使用 `npm` 进行包管理，所有依赖、配置、启动信息等会放到 `package.json` 文件里。在拿到程序代码后，需要先进行 `npm install` 才可以获得所有需要的依赖。然后就可以通过 `npm start` 来启动应用。因此，一般来说会这样写 `Dockerfile`：

```
FROM node:slim
RUN mkdir /app
WORKDIR /app
COPY ./package.json /app
RUN [ "npm", "install" ]
COPY . /app/
CMD [ "npm", "start" ]
```

把这个 `Dockerfile` 放到 Node.js 项目的根目录，构建好镜像后，就可以直接拿来启动容器运行。但是如果我们还有第二个 Node.js 项目也差不多呢？好吧，那就再把这个 `Dockerfile` 复制到第二个项目里。那如果有第三个项目呢？再复制么？文件的副本越多，版本控制就越困难，让我们继续看这样的场景维护的问题。

如果第一个 Node.js 项目在开发过程中，发现这个 `Dockerfile` 里存在问题，比如敲错字了、或者需要安装额外的包，然后开发人员修复了这个 `Dockerfile`，再次构建，问题解决。第一个项目没问题了，但是第二个项目呢？虽然最初 `Dockerfile` 是复制、粘贴自第一个项目的，但是并不会因为第一个项目修复了他们的 `Dockerfile`，而第二个项目的 `Dockerfile` 就会被自动修复。

那么我们可不可以做一个基础镜像，然后各个项目使用这个基础镜像呢？这样基础镜像更新，各个项目不用同步 `Dockerfile` 的变化，重新构建后就继承了基础镜像的更新？好吧，可以，让我们看看这样的结果。那么上面的这个 `Dockerfile` 就会变为：

```
FROM node:slim
RUN mkdir /app
WORKDIR /app
CMD [ "npm", "start" ]
```

这里我们把项目相关的构建指令拿出来，放到子项目里去。假设这个基础镜像的名字为 `my-node` 的话，各个项目内的自己的 `Dockerfile` 就变为：

```
FROM my-node
COPY ./package.json /app
RUN [ "npm", "install" ]
COPY . /app/
```

基础镜像变化后，各个项目都用这个 `Dockerfile` 重新构建镜像，会继承基础镜像的更新。

那么，问题解决了么？没有。准确说，只解决了一半。如果这个 `Dockerfile` 里面有些东西需要调整呢？比如 `npm install` 都需要加一些参数，那怎么办？这一行 `RUN` 是不可能放入基础镜像的，因为涉及到了当前项目的 `./package.json`，难道又要一个个修改么？所以说，这样制作基础镜像，只解决了原来的 `Dockerfile` 的前4条指令的变化问题，而后面三条指令的变化则完全没办法处理。

`ONBUILD` 可以解决这个问题。让我们用 `ONBUILD` 重新写一下基础镜像的 `Dockerfile`：

```
FROM node:slim
RUN mkdir /app
WORKDIR /app
ONBUILD COPY ./package.json /app
ONBUILD RUN [ "npm", "install" ]
ONBUILD COPY . /app/
CMD [ "npm", "start" ]
```

这次我们回到原始的 `Dockerfile`，但是这次将项目相关的指令加上 `ONBUILD`，这样在构建基础镜像的时候，这三行并不会被执行。然后各个项目的 `Dockerfile` 就变成了简单地：

```
FROM my-node
```

是的，只有这么一行。当在各个项目目录中，用这个只有一行的 `Dockerfile` 构建镜像时，之前基础镜像的那三行 `ONBUILD` 就会开始执行，成功的将当前项目的代码复制进镜像、并且针对本项目执行 `npm install`，生成应用镜像。

参考文档

* Dockerfie 官方文档：https://docs.docker.com/engine/reference/builder/
* Dockerfile 最佳实践文档：https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

其它生成镜像的方法

除了标准的使用 `Dockerfile` 生成镜像的方法外，由于各种特殊需求和历史原因，还提供了一些其它方法用以生成镜像。

从 rootfs 压缩包导入

格式：`docker import [选项] <文件>|<URL>|- [<仓库名>[:<标签>]]`

压缩包可以是本地文件、远程 Web 文件，甚至是从标准输入中得到。压缩包将会在镜像 `/` 目录展开，并直接作为镜像第一层提交。

比如我们想要创建一个 OpenVZ 的 Ubuntu 14.04 模板的镜像：

```
$ docker import \
    http://download.openvz.org/template/precreated/ubuntu-14.04-
x86_64-minimal.tar.gz \
    openvz/ubuntu:14.04
Downloading from http://download.openvz.org/template/precreated/
ubuntu-14.04-x86_64-minimal.tar.gz
sha256:f477a6e18e989839d25223f301ef738b69621c4877600ae6467c4e528
9822a79B/78.42 MB
```

这条命令自动下载了 `ubuntu-14.04-x86_64-minimal.tar.gz` 文件，并且作为根文件系统展开导入，并保存为镜像 `openvz/ubuntu:14.04` 。

导入成功后，我们可以用 `docker images` 看到这个导入的镜像：

```
$ docker images openvz/ubuntu
REPOSITORY          TAG                 IMAGE ID            CREA
TED             SIZE
openvz/ubuntu       14.04               f477a6e18e98        55 s
econds ago      214.9 MB
```

如果我们查看其历史的话，会看到描述中有导入的文件链接：

```
$ docker history openvz/ubuntu:14.04
IMAGE               CREATED             CREATED BY          SIZ
E                   COMMENT
f477a6e18e98        About a minute ago                      214
.9 MB               Imported from http://download.openvz.org/templa
te/precreated/ubuntu-14.04-x86_64-minimal.tar.gz
```

`docker save` 和 `docker load`

Docker 还提供了 `docker load` 和 `docker save` 命令，用以将镜像保存为一个 `tar` 文件，然后传输到另一个位置上，再加载进来。这是在没有 Docker Registry 时的做法，现在已经不推荐，镜像迁移应该直接使用 Docker Registry，无论是直接使用 Docker Hub 还是使用内网私有 Registry 都可以。

保存镜像

使用 `docker save` 命令可以将镜像保存为归档文件。

比如我们希望保存这个 `alpine` 镜像。

```
$ docker images alpine
REPOSITORY          TAG                 IMAGE ID            CREA
TED             SIZE
alpine              latest              baa5d63471ea        5 we
eks ago         4.803 MB
```

保存镜像的命令为：

```
$ docker save alpine | gzip > alpine-latest.tar.gz
```

然后我们将 `alpine-latest.tar.gz` 文件复制到了到了另一个机器上，可以用下面这个命令加载镜像：

```
$ docker load -i alpine-latest.tar.gz
Loaded image: alpine:latest
```

如果我们结合这两个命令以及 `ssh` 甚至 `pv` 的话，利用 Linux 强大的管道，我们可以写一个命令完成从一个机器将镜像迁移到另一个机器，并且带进度条的功能：

```
docker save <镜像名> | bzip2 | pv | ssh <用户名>@<主机名> 'cat | docker load'
```

删除本地镜像

如果要删除本地的镜像，可以使用 `docker rmi` 命令，其格式为：

```
docker rmi [选项] <镜像1> [<镜像2> ...]
```

注意 `docker rm` 命令是删除容器，不要混淆。

用 ID、镜像名、摘要删除镜像

其中，`<镜像>` 可以是 `镜像短 ID`、`镜像长 ID`、`镜像名` 或者 `镜像摘要`。

比如我们有这么一些镜像：

```
$ docker images
REPOSITORY                    TAG                 IMAGE ID
    CREATED             SIZE
centos                        latest              0584b3d2cf6d
    3 weeks ago         196.5 MB
redis                         alpine              501ad78535f0
    3 weeks ago         21.03 MB
docker                        latest              cf693ec9b5c7
    3 weeks ago         105.1 MB
nginx                         latest              e43d811ce2f4
    5 weeks ago         181.5 MB
```

我们可以用镜像的完整 ID，也称为 `长 ID`，来删除镜像。使用脚本的时候可能会用长 ID，但是人工输入就太累了，所以更多的时候是用 `短 ID` 来删除镜像。`docker images` 默认列出的就已经是短 ID 了，一般取前3个字符以上，只要足够区分于别的镜像就可以了。

比如这里，如果我们要删除 `redis:alpine` 镜像，可以执行：

```
$ docker rmi 501
Untagged: redis:alpine
Untagged: redis@sha256:f1ed3708f538b537eb9c2a7dd50dc90a706f7debd7e1196c9264edeea521a86d
Deleted: sha256:501ad78535f015d88872e13fa87a828425117e3d28075d0c117932b05bf189b7
Deleted: sha256:96167737e29ca8e9d74982ef2a0dda76ed7b430da55e321c071f0dbff8c2899b
Deleted: sha256:32770d1dcf835f192cafd6b9263b7b597a1778a403a109e2cc2ee866f74adf23
Deleted: sha256:127227698ad74a5846ff5153475e03439d96d4b1c7f2a449c7a826ef74a2d2fa
Deleted: sha256:1333ecc582459bac54e1437335c0816bc17634e131ea0cc48daa27d32c75eab3
Deleted: sha256:4fc455b921edf9c4aea207c51ab39b10b06540c8b4825ba57b3feed1668fa7c7
```

我们也可以用 `镜像名` ，也就是 `<仓库名>:<标签>` ，来删除镜像。

```
$ docker rmi centos
Untagged: centos:latest
Untagged: centos@sha256:b2f9d1c0ff5f87a4743104d099a3d561002ac500db1b9bfa02a783a46e0d366c
Deleted: sha256:0584b3d2cf6d235ee310cf14b54667d889887b838d3f3d3033acd70fc3c48b8a
Deleted: sha256:97ca462ad9eeae25941546209454496e1d66749d53dfa2ee32bf1faabd239d38
```

当然，更精确的是使用 `镜像摘要` 删除镜像。

```
$ docker images --digests
REPOSITORY                  TAG                 DIGEST
                                                          IMAGE
ID              CREATED             SIZE
node                        slim                sha256:b4f0e0bde
b578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228   6e0c4c
8e3913          3 weeks ago         214 MB

$ docker rmi node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4
a582f3be235a3b164422be228
Untagged: node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a58
2f3be235a3b164422be228
```

Untagged 和 Deleted

如果观察上面这几个命令的运行输出信息的话，你会注意到删除行为分为两类，一类是 `Untagged` ，另一类是 `Deleted` 。我们之前介绍过，镜像的唯一标识是其 ID 和摘要，而一个镜像可以有多个标签。

因此当我们使用上面命令删除镜像的时候，实际上是在要求删除某个标签的镜像。所以首先需要做的是将满足我们要求的所有镜像标签都取消，这就是我们看到的 `Untagged` 的信息。因为一个镜像可以对应多个标签，因此当我们删除了所指定的标签后，可能还有别的标签指向了这个镜像，如果是这种情况，那么 `Delete` 行为就不会发生。所以并非所有的 `docker rmi` 都会产生删除镜像的行为，有可能仅仅是取消了某个标签而已。

当该镜像所有的标签都被取消了，该镜像很可能会失去了存在的意义，因此会触发删除行为。镜像是多层存储结构，因此在删除的时候也是从上层向基础层方向依次进行判断删除。镜像的多层结构让镜像复用变动非常容易，因此很有可能某个其它镜像正依赖于当前镜像的某一层。这种情况，依旧不会触发删除该层的行为。直到没有任何层依赖当前层时，才会真实的删除当前层。这就是为什么，有时候会奇怪，为什么明明没有别的标签指向这个镜像，但是它还是存在的原因，也是为什么有时候会发现所删除的层数和自己 `docker pull` 看到的层数不一样的源。

除了镜像依赖以外，还需要注意的是容器对镜像的依赖。如果有用这个镜像启动的容器存在（即使容器没有运行），那么同样不可以删除这个镜像。之前讲过，容器是以镜像为基础，再加一层容器存储层，组成这样的多层存储结构去运行的。因此

该镜像如果被这个容器所依赖的，那么删除必然会导致故障。如果这些容器是不需要的，应该先将它们删除，然后再来删除镜像。

用 docker images 命令来配合

像其它可以承接多个实体的命令一样，可以使用 `docker images -q` 来配合使用 `docker rmi`，这样可以成批的删除希望删除的镜像。比如之前我们介绍过的，删除虚悬镜像的指令是：

```
$ docker rmi $(docker images -q -f dangling=true)
```

我们在“镜像列表”章节介绍过很多过滤镜像列表的方式都可以拿过来使用。

比如，我们需要删除所有仓库名为 `redis` 的镜像：

```
$ docker rmi $(docker images -q redis)
```

或者删除所有在 `mongo:3.2` 之前的镜像：

```
$ docker rmi $(docker images -q -f before=mongo:3.2)
```

充分利用你的想象力和 Linux 命令行的强大，你可以完成很多非常赞的功能。

CentOS/RHEL 的用户需要注意的事项

在 Ubuntu/Debian 上有 `UnionFS` 可以使用，如 `aufs` 或者 `overlay2`，而 CentOS 和 RHEL 的内核中没有相关驱动。因此对于这类系统，一般使用 `devicemapper` 驱动利用 LVM 的一些机制来模拟分层存储。这样的做法除了性能比较差外，稳定性一般也不好，而且配置相对复杂。Docker 安装在 CentOS/RHEL 上后，会默认选择 `devicemapper`，但是为了简化配置，其 `devicemapper` 是跑在一个稀疏文件模拟的块设备上，也被称为 `loop-lvm`。这样的选择是因为不需要额外配置就可以运行 Docker，这是自动配置唯一能做到的事情。但是 `loop-lvm` 的做法非常不好，其稳定性、性能更差，无论是日志还是 `docker info` 中都会看到警告信息。官方文档有明确的文章讲解了如何配置块设备给 `devicemapper` 驱动做存储层的做法，这类做法也被称为配置 `direct-lvm`。

除了前面说到的问题外，`devicemapper` + `loop-lvm` 还有一个缺陷，因为它是稀疏文件，所以它会不断增长。用户在使用过程中会注意到 `/var/lib/docker/devicemapper/devicemapper/data` 不断增长，而且无法控制。很多人会希望删除镜像或者可以解决这个问题，结果发现效果并不明显。原因就是这个稀疏文件的空间释放后基本不进行垃圾回收的问题。因此往往会出现即使删除了文件内容，空间却无法回收，随着使用这个稀疏文件一直在不断增长。

所以对于 CentOS/RHEL 的用户来说，在没有办法使用 `UnionFS` 的情况下，一定要配置 `direct-lvm` 给 `devicemapper`，无论是为了性能、稳定性还是空间利用率。

或许有人注意到了 *CentOS 7* 中存在被 *backports* 回来的 `overlay` 驱动，不过 *CentOS* 里的这个驱动达不到生产环境使用的稳定程度，所以不推荐使用。

镜像的实现原理

Docker 镜像是怎么实现增量的修改和维护的？ 每个镜像都由很多层次构成，Docker 使用 Union FS 将这些不同的层结合到一个镜像中去。

通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个 disk 挂到同一个目录下,另一个更常用的就是将一个只读的分支和一个可写的分支联合在一起，Live CD 正是基于此方法可以允许在镜像不变的基础上允许用户在其上进行一些写操作。 Docker 在 AUFS 上构建的容器也是利用了类似的原理。

操作 Docker 容器

容器是 Docker 又一核心概念。

简单的说，容器是独立运行的一个或一组应用，以及它们的运行态环境。对应的，虚拟机可以理解为模拟运行的一整套操作系统（提供了运行态环境和其他系统环境）和跑在上面的应用。

本章将具体介绍如何来管理一个容器，包括创建、启动和停止等。

启动容器

启动容器有两种方式，一种是基于镜像新建一个容器并启动，另外一个是将在终止状态（stopped）的容器重新启动。

因为 Docker 的容器实在太轻量级了，很多时候用户都是随时删除和新创建容器。

新建并启动

所需要的命令主要为 `docker run`。

例如，下面的命令输出一个 “Hello World”，之后终止容器。

```
$ sudo docker run ubuntu:14.04 /bin/echo 'Hello world'
Hello world
```

这跟在本地直接执行 `/bin/echo 'hello world'` 几乎感觉不出任何区别。

下面的命令则启动一个 bash 终端，允许用户进行交互。

```
$ sudo docker run -t -i ubuntu:14.04 /bin/bash
root@af8bae53bdd3:/#
```

其中， `-t` 选项让Docker分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上， `-i` 则让容器的标准输入保持打开。

在交互模式下，用户可以通过所创建的终端来输入命令，例如

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin
 srv sys tmp usr var
```

当利用 `docker run` 来创建容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载

- 利用镜像创建并启动一个容器
- 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
- 从地址池配置一个 ip 地址给容器
- 执行用户指定的应用程序
- 执行完毕后容器被终止

启动已终止容器

可以利用 `docker start` 命令，直接将一个已经终止的容器启动运行。

容器的核心为所执行的应用程序，所需要的资源都是应用程序运行所必需的。除此之外，并没有其它的资源。可以在伪终端中利用 `ps` 或 `top` 来查看进程信息。

```
root@ba267838cc1b:/# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   11 ?        00:00:00 ps
```

可见，容器中仅运行了指定的 bash 应用。这种特点使得 Docker 对资源的利用率极高，是货真价实的轻量级虚拟化。

后台(background)运行

更多的时候，需要让 Docker在后台运行而不是直接把执行命令的结果输出在当前宿主机下。此时，可以通过添加 `-d` 参数来实现。

下面举两个例子来说明一下。

如果不使用 `-d` 参数运行容器。

```
$ sudo docker run ubuntu:14.04 /bin/sh -c "while true; do echo h
ello world; sleep 1; done"
hello world
hello world
hello world
hello world
```

容器会把输出的结果(STDOUT)打印到宿主机上面

如果使用了 `-d` 参数运行容器。

```
$ sudo docker run -d ubuntu:14.04 /bin/sh -c "while true; do ech
o hello world; sleep 1; done"
77b2dc01fe0f3f1265df143181e7b9af5e05279a884f4776ee75350ea9d8017a
```

此时容器会在后台运行并不会把输出的结果(STDOUT)打印到宿主机上面(输出结果可以用docker logs 查看)。

注： 容器是否会长久运行，是和docker run指定的命令有关，和 `-d` 参数无关。

使用 `-d` 参数启动后会返回一个唯一的 id，也可以通过 `docker ps` 命令来查看容器信息。

```
$ sudo docker ps
CONTAINER ID  IMAGE         COMMAND               CREATED
 STATUS       PORTS NAMES
77b2dc01fe0f  ubuntu:14.04  /bin/sh -c 'while tr  2 minutes ago
 Up 1 minute        agitated_wright
```

要获取容器的输出信息，可以通过 `docker logs` 命令。

```
$ sudo docker logs [container ID or NAMES]
hello world
hello world
hello world
. . .
```

终止容器

可以使用 `docker stop` 来终止一个运行中的容器。

此外，当Docker容器中指定的应用终结时，容器也自动终止。 例如对于上一章节中只启动了一个终端的容器，用户通过 `exit` 命令或 `Ctrl+d` 来退出终端时，所创建的容器立刻终止。

终止状态的容器可以用 `docker ps -a` 命令看到。例如

```
sudo docker ps -a
CONTAINER ID        IMAGE                    COMMAND
    CREATED             STATUS                          PORTS
            NAMES
ba267838cc1b        ubuntu:14.04             "/bin/bash"
    30 minutes ago      Exited (0) About a minute ago
            trusting_newton
98e5efa7d997        training/webapp:latest   "python app.py"
    About an hour ago   Exited (0) 34 minutes ago
            backstabbing_pike
```

处于终止状态的容器，可以通过 `docker start` 命令来重新启动。

此外， `docker restart` 命令会将一个运行态的容器终止，然后再重新启动它。

进入容器

在使用 `-d` 参数时，容器启动后会进入后台。 某些时候需要进入容器进行操作，有很多种方法，包括使用 `docker attach` 命令或 `nsenter` 工具等。

attach 命令

`docker attach` 是Docker自带的命令。下面示例如何使用该命令。

```
$ sudo docker run -idt ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
TED             STATUS              PORTS               NAMES
243c32535da7        ubuntu:latest       "/bin/bash"         18 s
econds ago      Up 17 seconds                           nostalgi
c_hypatia
$sudo docker attach nostalgic_hypatia
root@243c32535da7:/#
```

但是使用 `attach` 命令有时候并不方便。当多个窗口同时 attach 到同一个容器的时候，所有窗口都会同步显示。当某个窗口因命令阻塞时,其他窗口也无法执行操作了。

nsenter 命令

安装

`nsenter` 工具在 util-linux 包2.23版本后包含。 如果系统中 util-linux 包没有该命令，可以按照下面的方法从源码安装。

```
$ cd /tmp; curl https://www.kernel.org/pub/linux/utils/util-linu
x/v2.24/util-linux-2.24.tar.gz | tar -zxf-; cd util-linux-2.24;
$ ./configure --without-ncurses
$ make nsenter && sudo cp nsenter /usr/local/bin
```

使用

`nsenter` 启动一个新的shell进程(默认是/bin/bash), 同时会把这个新进程切换到和目标(target)进程相同的命名空间，这样就相当于进入了容器内部。nsenter 要正常工作需要有 root 权限。 很不幸，Ubuntu 14.04 仍然使用的是 util-linux 2.20。安装最新版本的 util-linux（2.29）版，请按照以下步骤：

```
$ wget https://www.kernel.org/pub/linux/utils/util-linux/v2.29/u
til-linux-2.29.tar.xz; tar xJvf util-linux-2.29.tar.xz
$ cd util-linux-2.29
$ ./configure --without-ncurses && make nsenter
$ sudo cp nsenter /usr/local/bin
```

为了连接到容器，你还需要找到容器的第一个进程的 PID，可以通过下面的命令获取。

```
PID=$(docker inspect --format "{{ .State.Pid }}" <container>)
```

通过这个 PID，就可以连接到这个容器：

```
$ nsenter --target $PID --mount --uts --ipc --net --pid
```

如果无法通过以上命令连接到这个容器，有可能是因为宿主的默认shell在容器中并不存在，比如zsh，可以使用如下命令显式地使用bash。

```
$ nsenter --target $pid --mount --uts --ipc --net --pid  -- /usr
/bin/env \
--ignore-environment HOME=/root /bin/bash --login
```

下面给出一个完整的例子。

```
$ sudo docker run -idt ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
TED             STATUS              PORTS               NAMES
243c32535da7        ubuntu:latest       "/bin/bash"         18 s
econds ago      Up 17 seconds                           nostalgi
c_hypatia
$ PID=$(docker-pid 243c32535da7)
10981
$ sudo nsenter --target 10981 --mount --uts --ipc --net --pid
root@243c32535da7:/#
```

更简单的，建议大家下载 .bashrc_docker，并将内容放到 .bashrc 中。

```
$ wget -P ~ https://github.com/yeasy/docker_practice/raw/master/
_local/.bashrc_docker;
$ echo "[ -f ~/.bashrc_docker ] && . ~/.bashrc_docker" >> ~/.bas
hrc; source ~/.bashrc
```

这个文件中定义了很多方便使用 Docker 的命令，例如 `docker-pid` 可以获取某个容器的 PID；而 `docker-enter` 可以进入容器或直接在容器内执行命令。

```
$ echo $(docker-pid <container>)
$ docker-enter <container> ls
```

导出和导入容器

导出容器

如果要导出本地某个容器，可以使用 `docker export` 命令。

```
$ sudo docker ps -a
CONTAINER ID        IMAGE                   COMMAND              CREA
TED              STATUS                    PORTS               NA
MES
7691a814370e        ubuntu:14.04        "/bin/bash"          36 h
ours ago         Exited (0) 21 hours ago                       te
st
$ sudo docker export 7691a814370e > ubuntu.tar
```

这样将导出容器快照到本地文件。

导入容器快照

可以使用 `docker import` 从容器快照文件中再导入为镜像，例如

```
$ cat ubuntu.tar | sudo docker import - test/ubuntu:v1.0
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREA
TED               VIRTUAL SIZE
test/ubuntu         v1.0                9d37a6082e97        Abou
t a minute ago    171.3 MB
```

此外，也可以通过指定 URL 或者某个目录来导入，例如

```
$sudo docker import http://example.com/exampleimage.tgz example/
imagerepo
```

*注：用户既可以使用 `docker load` 来导入镜像存储文件到本地镜像库，也可以使用 `docker import` 来导入一个容器快照到本地镜像库。这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息（即仅保存容器当时的快照状态），而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

删除容器

可以使用 `docker rm` 来删除一个处于终止状态的容器。 例如

```
$sudo docker rm  trusting_newton
trusting_newton
```

如果要删除一个运行中的容器，可以添加 `-f` 参数。Docker 会发送 `SIGKILL` 信号给容器。

清理所有处于终止状态的容器

用 `docker ps -a` 命令可以查看所有已经创建的包括终止状态的容器，如果数量太多要一个个删除可能会很麻烦，用 `docker rm $(docker ps -a -q)` 可以全部清理掉。

*注意：这个命令其实会试图删除所有的包括还在运行中的容器，不过就像上面提过的 `docker rm` 默认并不会删除运行中的容器。

访问仓库

仓库（Repository）是集中存放镜像的地方。

一个容易混淆的概念是注册服务器（Registry）。实际上注册服务器是管理仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像。从这方面来说，仓库可以被认为是一个具体的项目或目录。例如对于仓库地址 `dl.dockerpool.com/ubuntu` 来说，`dl.dockerpool.com` 是注册服务器地址，`ubuntu` 是仓库名。

大部分时候，并不需要严格区分这两者的概念。

Docker Hub

目前 Docker 官方维护了一个公共仓库 Docker Hub，其中已经包括了超过 15,000 的镜像。大部分需求，都可以通过在 Docker Hub 中直接下载镜像来实现。

登录

可以通过执行 `docker login` 命令来输入用户名、密码和邮箱来完成注册和登录。 注册成功后，本地用户目录的 `.dockercfg` 中将保存用户的认证信息。

基本操作

用户无需登录即可通过 `docker search` 命令来查找官方仓库中的镜像，并利用 `docker pull` 命令来将它下载到本地。

例如以 centos 为关键词进行搜索：

```
$ sudo docker search centos
NAME                                                DESCRIPTION
                                STARS     OFFICIAL   AUTOMATED
centos                                              The official bui
ld of CentOS.                   465       [OK]
tianon/centos                                       CentOS 5 and 6,
created using rinse instea...   28
blalor/centos                                       Bare-bones base
CentOS 6.5 image                6                    [OK]
saltstack/centos-6-minimal
                                6                    [OK]
tutum/centos-6.4                                    DEPRECATED. Use
tutum/centos:6.4 instead. ...   5                    [OK]
...
```

可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、星级（表示该镜像的受欢迎程度）、是否官方创建、是否自动创建。 官方的镜像说明是官方项目组创建和维护的，automated 资源允许用户验证镜像的来源和内容。

根据是否是官方提供，可将镜像资源分为两类。 一种是类似 centos 这样的基础镜像，被称为基础或根镜像。这些基础镜像是由 Docker 公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。 还有一种类型，比如 `tianon/centos` 镜像，它是由 Docker 的用户创建并维护的，往往带有用户名称前缀。可以通过前缀 `user_name/` 来指定使用某个用户提供的镜像，比如 tianon 用户。

另外，在查找的时候通过 `-s N` 参数可以指定仅显示评价为 `N` 星以上的镜像（新版本Docker推荐使用 `--filter=stars=N` 参数）。

下载官方 centos 镜像到本地。

```
$ sudo docker pull centos
Pulling repository centos
0b443ba03958: Download complete
539c0211cd76: Download complete
511136ea3c5a: Download complete
7064731afe90: Download complete
```

用户也可以在登录后通过 `docker push` 命令来将镜像推送到 Docker Hub。

自动创建

自动创建（Automated Builds）功能对于需要经常升级镜像内程序来说，十分方便。 有时候，用户创建了镜像，安装了某个软件，如果软件发布新版本则需要手动更新镜像。。

而自动创建允许用户通过 Docker Hub 指定跟踪一个目标网站（目前支持 GitHub 或 BitBucket）上的项目，一旦项目发生新的提交，则自动执行创建。

要配置自动创建，包括如下的步骤：

- 创建并登录 Docker Hub，以及目标网站；
- 在目标网站中连接帐户到 Docker Hub；
- 在 Docker Hub 中 配置一个自动创建；
- 选取一个目标网站中的项目（需要含 Dockerfile）和分支；
- 指定 Dockerfile 的位置，并提交创建。

之后，可以 在Docker Hub 的 自动创建页面 中跟踪每次创建的状态。

私有仓库

有时候使用 Docker Hub 这样的公共仓库可能不方便，用户可以创建一个本地仓库供私人使用。

本节介绍如何使用本地仓库。

`docker-registry` 是官方提供的工具，可以用于构建私有的镜像仓库。

安装运行 docker-registry

容器运行

在安装了 Docker 后，可以通过获取官方 registry 镜像来运行。

```
$ sudo docker run -d -p 5000:5000 registry
```

这将使用官方的 registry 镜像来启动本地的私有仓库。 用户可以通过指定参数来配置私有仓库位置，例如配置镜像存储到 Amazon S3 服务。

```
$ sudo docker run \
         -e SETTINGS_FLAVOR=s3 \
         -e AWS_BUCKET=acme-docker \
         -e STORAGE_PATH=/registry \
         -e AWS_KEY=AKIAHSHB43HS3J92MXZ \
         -e AWS_SECRET=xdDowwlK7TJajV1Y7EoOZrmuPEJlHYcNP2k4j49T
\
         -e SEARCH_BACKEND=sqlalchemy \
         -p 5000:5000 \
         registry
`
```

此外，还可以指定本地路径（如 `/home/user/registry-conf` ）下的配置文件。

```
$ sudo docker run -d -p 5000:5000 -v /home/user/registry-conf:/registry-conf -e DOCKER_REGISTRY_CONFIG=/registry-conf/config.yml registry
```

默认情况下，仓库会被创建在容器的 `/var/lib/registry` （v1 中是 `/tmp/registry` ）下。可以通过 `-v` 参数来将镜像文件存放在本地的指定路径。例如下面的例子将上传的镜像放到 `/opt/data/registry` 目录。

```
$ sudo docker run -d -p 5000:5000 -v /opt/data/registry:/var/lib/registry registry
```

本地安装

对于 Ubuntu 或 CentOS 等发行版，可以直接通过源安装。

- Ubuntu

  ```
  $ sudo apt-get install -y build-essential python-dev libevent-dev python-pip liblzma-dev
  $ sudo pip install docker-registry
  ```

- CentOS

  ```
  $ sudo yum install -y python-devel libevent-devel python-pip gcc xz-devel
  $ sudo python-pip install docker-registry
  ```

也可以从 docker-registry 项目下载源码进行安装。

```
$ sudo apt-get install build-essential python-dev libevent-dev python-pip libssl-dev liblzma-dev libffi-dev
$ git clone https://github.com/docker/distribution
$ cd distribution
$ sudo docker build .
```

启动运行registry的容器

```
$ sudo docker run -d -p 5000:5000 --restart=always --name regist
ry ${IMAGE_ID}
```

在私有仓库上传、下载、搜索镜像

创建好私有仓库之后，就可以使用 `docker tag` 来标记一个镜像，然后推送它到仓库，别的机器上就可以下载下来了。例如私有仓库地址为 `192.168.7.26:5000` 。

先在本机查看已有的镜像。

```
$ sudo docker images
REPOSITORY                        TAG                 IMAGE ID
          CREATED             VIRTUAL SIZE
ubuntu                            latest              ba5877dc9b
ec        6 weeks ago         192.7 MB
ubuntu                            14.04               ba5877dc9b
ec        6 weeks ago         192.7 MB
```

使用 `docker tag` 将 `ba58` 这个镜像标记为 `192.168.7.26:5000/test` （格式为 `docker tag IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]` ）。

```
$ sudo docker tag ba58 192.168.7.26:5000/test
root ~ # docker images
REPOSITORY                        TAG                 IMAGE ID
          CREATED             VIRTUAL SIZE
ubuntu                            14.04               ba5877dc9b
ec        6 weeks ago         192.7 MB
ubuntu                            latest              ba5877dc9b
ec        6 weeks ago         192.7 MB
192.168.7.26:5000/test            latest              ba5877dc9b
ec        6 weeks ago         192.7 MB
```

使用 `docker push` 上传标记的镜像。

```
$ sudo docker push 192.168.7.26:5000/test
The push refers to a repository [192.168.7.26:5000/test] (len: 1
)
Sending image list
Pushing repository 192.168.7.26:5000/test (1 tags)
Image 511136ea3c5a already pushed, skipping
Image 9bad880da3d2 already pushed, skipping
Image 25f11f5fb0cb already pushed, skipping
Image ebc34468f71d already pushed, skipping
Image 2318d26665ef already pushed, skipping
Image ba5877dc9bec already pushed, skipping
Pushing tag for rev [ba5877dc9bec] on {http://192.168.7.26:5000/
v1/repositories/test/tags/latest}
```

用 curl 查看仓库中的镜像。

```
$ curl http://192.168.7.26:5000/v1/search
{"num_results": 7, "query": "", "results": [{"description": "",
"name": "library/miaxis_j2ee"}, {"description": "", "name": "lib
rary/tomcat"}, {"description": "", "name": "library/ubuntu"}, {"
description": "", "name": "library/ubuntu_office"}, {"descriptio
n": "", "name": "library/desktop_ubu"}, {"description": "", "nam
e": "dockerfile/ubuntu"}, {"description": "", "name": "library/t
est"}]}
```

这里可以看到 `{"description": "", "name": "library/test"}`，表明镜像已经被成功上传了。

现在可以到另外一台机器去下载这个镜像。

```
$ sudo docker pull 192.168.7.26:5000/test
Pulling repository 192.168.7.26:5000/test
ba5877dc9bec: Download complete
511136ea3c5a: Download complete
9bad880da3d2: Download complete
25f11f5fb0cb: Download complete
ebc34468f71d: Download complete
2318d26665ef: Download complete
$ sudo docker images
REPOSITORY                        TAG                 IMAGE ID
           CREATED              VIRTUAL SIZE
192.168.7.26:5000/test                latest              ba5877dc9
bec          6 weeks ago          192.7 MB
```

可以使用 这个脚本 批量上传本地的镜像到注册服务器中，默认是本地注册服务器 `127.0.0.1:5000` 。例如：

```
$ wget https://github.com/yeasy/docker_practice/raw/master/_local/push_images.sh; sudo chmod a+x push_images.sh
$ ./push_images.sh ubuntu:latest centos:centos7
The registry server is 127.0.0.1
Uploading ubuntu:latest...
The push refers to a repository [127.0.0.1:5000/ubuntu] (len: 1)
Sending image list
Pushing repository 127.0.0.1:5000/ubuntu (1 tags)
Image 511136ea3c5a already pushed, skipping
Image bfb8b5a2ad34 already pushed, skipping
Image c1f3bdbd8355 already pushed, skipping
Image 897578f527ae already pushed, skipping
Image 9387bcc9826e already pushed, skipping
Image 809ed259f845 already pushed, skipping
Image 96864a7d2df3 already pushed, skipping
Pushing tag for rev [96864a7d2df3] on {http://127.0.0.1:5000/v1/repositories/ubuntu/tags/latest}
Untagged: 127.0.0.1:5000/ubuntu:latest
Done
Uploading centos:centos7...
The push refers to a repository [127.0.0.1:5000/centos] (len: 1)
Sending image list
Pushing repository 127.0.0.1:5000/centos (1 tags)
Image 511136ea3c5a already pushed, skipping
34e94e67e63a: Image successfully pushed
70214e5d0a90: Image successfully pushed
Pushing tag for rev [70214e5d0a90] on {http://127.0.0.1:5000/v1/repositories/centos/tags/centos7}
Untagged: 127.0.0.1:5000/centos:centos7
Done
```

仓库配置文件

Docker 的 Registry 利用配置文件提供了一些仓库的模板（flavor），用户可以直接使用它们来进行开发或生产部署。

模板

在 `config_sample.yml` 文件中，可以看到一些现成的模板段：

- `common` ：基础配置
- `local` ：存储数据到本地文件系统
- `s3` ：存储数据到 AWS S3 中
- `dev` ：使用 `local` 模板的基本配置
- `test` ：单元测试使用
- `prod` ：生产环境配置（基本上跟s3配置类似）
- `gcs` ：存储数据到 Google 的云存储
- `swift` ：存储数据到 OpenStack Swift 服务
- `glance` ：存储数据到 OpenStack Glance 服务，本地文件系统为后备
- `glance-swift` ：存储数据到 OpenStack Glance 服务，Swift 为后备
- `elliptics` ：存储数据到 Elliptics key/value 存储

用户也可以添加自定义的模版段。

默认情况下使用的模板是 `dev` ，要使用某个模板作为默认值，可以添加 `SETTINGS_FLAVOR` 到环境变量中，例如

```
export SETTINGS_FLAVOR=dev
```

另外，配置文件中支持从环境变量中加载值，语法格式为 `_env:VARIABLENAME[:DEFAULT]` 。

示例配置

```
common:
    loglevel: info
    search_backend: "_env:SEARCH_BACKEND:"
    sqlalchemy_index_database:
        "_env:SQLALCHEMY_INDEX_DATABASE:sqlite:////tmp/docker-re
gistry.db"

prod:
    loglevel: warn
    storage: s3
    s3_access_key: _env:AWS_S3_ACCESS_KEY
    s3_secret_key: _env:AWS_S3_SECRET_KEY
    s3_bucket: _env:AWS_S3_BUCKET
    boto_bucket: _env:AWS_S3_BUCKET
    storage_path: /srv/docker
    smtp_host: localhost
    from_addr: docker@myself.com
    to_addr: my@myself.com

dev:
    loglevel: debug
    storage: local
    storage_path: /home/myself/docker

test:
    storage: local
    storage_path: /tmp/tmpdockertmp
```

选项

Docker 数据管理

这一章介绍如何在 Docker 内部以及容器之间管理数据，在容器中管理数据主要有两种方式：

- 数据卷（Data volumes）
- 数据卷容器（Data volume containers）

数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

- 数据卷可以在容器之间共享和重用
- 对数据卷的修改会立马生效
- 对数据卷的更新，不会影响镜像
- 数据卷默认会一直存在，即使容器被删除

*注意：数据卷的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的数据卷。

创建一个数据卷

在用 `docker run` 命令的时候，使用 `-v` 标记来创建一个数据卷并挂载到容器里。在一次 run 中多次使用可以挂载多个数据卷。

下面创建一个名为 web 的容器，并加载一个数据卷到容器的 `/webapp` 目录。

```
$ sudo docker run -d -P --name web -v /webapp training/webapp py
thon app.py
```

*注意：也可以在 Dockerfile 中使用 `VOLUME` 来添加一个或者多个新的卷到由该镜像创建的任意容器。

删除数据卷

数据卷是被设计用来持久化数据的，它的生命周期独立于容器，Docker不会在容器被删除后自动删除数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的数据卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用 `docker rm -v` 这个命令。无主的数据卷可能会占据很多空间，要清理会很麻烦。Docker官方正在试图解决这个问题，相关工作的进度可以查看这个PR。

挂载一个主机目录作为数据卷

使用 `-v` 标记也可以指定挂载一个本地主机的目录到容器中去。

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp tr
aining/webapp python app.py
```

上面的命令加载主机的 `/src/webapp` 目录到容器的 `/opt/webapp` 目录。这个功能在进行测试的时候十分方便，比如用户可以放置一些程序到本地目录中，来查看容器是否正常工作。本地目录的路径必须是绝对路径，如果目录不存在 Docker 会自动为你创建它。

*注意：Dockerfile 中不支持这种用法，这是因为 Dockerfile 是为了移植和分享用的。然而，不同操作系统的路径格式不一样，所以目前还不能支持。

Docker 挂载数据卷的默认权限是读写，用户也可以通过 `:ro` 指定为只读。

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp:ro
training/webapp python app.py
```

加了 `:ro` 之后，就挂载为只读了。

查看数据卷的具体信息

在主机里使用以下命令可以查看指定容器的信息

```
$ docker inspect web
...
```

在输出的内容中找到其中和数据卷相关的部分，可以看到所有的数据卷都是创建在主机的 `/var/lib/docker/volumes/` 下面的

```
"Volumes": {
    "/webapp": "/var/lib/docker/volumes/fac362...80535"
},
"VolumesRW": {
    "/webapp": true
}
...
```

注：从Docker 1.8.0起，数据卷配置在"Mounts"Key下面，可以看到所有的数据卷都是创建在主机的 `/mnt/sda1/var/lib/docker/volumes/....` 下面了。

```
"Mounts": [
            {
                "Name": "b53ebd40054dae599faf7c9666acfe205c3e922
fc3e8bc3f2fd178ed788f1c29",
                "Source": "/mnt/sda1/var/lib/docker/volumes/b53e
bd40054dae599faf7c9666acfe205c3e922fc3e8bc3f2fd178ed788f1c29/_da
ta",
                "Destination": "/webapp",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            }
        ]
...
```

挂载一个本地主机文件作为数据卷

`-v` 标记也可以从主机挂载单个文件到容器中

```
$ sudo docker run --rm -it -v ~/.bash_history:/.bash_history ubu
ntu /bin/bash
```

这样就可以记录在容器输入过的命令了。

*注意：如果直接挂载一个文件，很多文件编辑工具，包括 `vi` 或者 `sed --in-place` ，可能会造成文件 inode 的改变，从 Docker 1.1 .0起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。

数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。

首先，创建一个名为 dbdata 的数据卷容器：

```
$ sudo docker run -d -v /dbdata --name dbdata training/postgres
echo Data-only container for postgres
```

然后，在其他容器中使用 `--volumes-from` 来挂载 dbdata 容器中的数据卷。

```
$ sudo docker run -d --volumes-from dbdata --name db1 training/p
ostgres
$ sudo docker run -d --volumes-from dbdata --name db2 training/p
ostgres
```

可以使用超过一个的 `--volumes-from` 参数来指定从多个容器挂载不同的数据卷。 也可以从其他已经挂载了数据卷的容器来级联挂载数据卷。

```
$ sudo docker run -d --name db3 --volumes-from db1 training/post
gres
```

*注意：使用 `--volumes-from` 参数所挂载数据卷的容器自己并不需要保持在运行状态。

如果删除了挂载的容器（包括 dbdata、db1 和 db2），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 `docker rm -v` 命令来指定同时删除关联的容器。 这可以让用户在容器之间升级和移动数据卷。具体的操作将在下一节中进行讲解。

利用数据卷容器来备份、恢复、迁移数据卷

可以利用数据卷对其中的数据进行进行备份、恢复和迁移。

备份

首先使用 `--volumes-from` 标记来创建一个加载 dbdata 容器卷的容器，并从主机挂载当前目录到容器的 /backup 目录。命令如下：

```
$ sudo docker run --volumes-from dbdata -v $(pwd):/backup ubuntu
 tar cvf /backup/backup.tar /dbdata
```

容器启动后，使用了 `tar` 命令来将 dbdata 卷备份为容器中 /backup/backup.tar 文件，也就是主机当前目录下的名为 `backup.tar` 的文件。

恢复

如果要恢复数据到一个容器，首先创建一个带有空数据卷的容器 dbdata2。

```
$ sudo docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

然后创建另一个容器，挂载 dbdata2 容器卷中的数据卷，并使用 `untar` 解压备份文件到挂载的容器卷中。

```
$ sudo docker run --volumes-from dbdata2 -v $(pwd):/backup busyb
ox tar xvf
/backup/backup.tar
```

为了查看/验证恢复的数据，可以再启动一个容器挂载同样的容器卷来查看

```
$ sudo docker run --volumes-from dbdata2 busybox /bin/ls /dbdata
```

Docker 中的网络功能介绍

Docker 允许通过外部访问容器或容器互联的方式来提供网络服务。

外部访问容器

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 `-P` 或 `-p` 参数来指定端口映射。

当使用 -P 标记时，Docker 会随机映射一个 `49000~49900` 的端口到内部容器开放的网络端口。

使用 `docker ps` 可以看到，本地主机的 49155 被映射到了容器的 5000 端口。此时访问本机的 49155 端口即可访问容器内 web 应用提供的界面。

```
$ sudo docker run -d -P training/webapp python app.py
$ sudo docker ps -l
CONTAINER ID  IMAGE                   COMMAND       CREATED
   STATUS        PORTS                    NAMES
bc533791f3f5  training/webapp:latest  python app.py 5 seconds ag
o  Up 2 seconds  0.0.0.0:49155->5000/tcp  nostalgic_morse
```

同样的，可以通过 `docker logs` 命令来查看应用的信息。

```
$ sudo docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1"
404 -
```

-p（小写的）则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。支持的格式有 `ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort` 。

映射所有接口地址

使用 `hostPort:containerPort` 格式本地的 5000 端口映射到容器的 5000 端口，可以执行

```
$ sudo docker run -d -p 5000:5000 training/webapp python app.py
```

此时默认会绑定本地所有接口上的所有地址。

映射到指定地址的指定端口

可以使用 `ip:hostPort:containerPort` 格式指定映射使用一个特定地址，比如 localhost 地址 127.0.0.1

```
$ sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp pyth
on app.py
```

映射到指定地址的任意端口

使用 `ip::containerPort` 绑定 localhost 的任意端口到容器的 5000 端口，本地主机会自动分配一个端口。

```
$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python a
pp.py
```

还可以使用 udp 标记来指定 udp 端口

```
$ sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp
python app.py
```

查看映射端口配置

使用 `docker port` 来查看当前映射的端口配置，也可以查看到绑定的地址

```
$ docker port nostalgic_morse 5000
127.0.0.1:49155.
```

注意：

- 容器有自己的内部网络和 ip 地址（使用 `docker inspect` 可以获取所有的变量，Docker 还可以有一个可变的网络配置。）
- -p 标记可以多次使用来绑定多个端口

例如

```
$ sudo docker run -d -p 5000:5000  -p 3000:80 training/webapp py
thon app.py
```

容器互联

容器的连接（linking）系统是除了端口映射外，另一种跟容器中应用交互的方式。

该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

自定义容器命名

连接系统依据容器的名称来执行。因此，首先需要自定义一个好记的容器命名。

虽然当创建容器的时候，系统默认会分配一个名字。自定义命名容器有2个好处：

- 自定义的命名，比较好记，比如一个web应用容器我们可以给它起名叫web
- 当要连接其他容器时候，可以作为一个有用的参考点，比如连接web容器到db容器

使用 `--name` 标记可以为容器自定义命名。

```
$ sudo docker run -d -P --name web training/webapp python app.py
```

使用 `docker ps` 来验证设定的命名。

```
$ sudo docker ps -l
CONTAINER ID  IMAGE                  COMMAND        CREATED
  STATUS       PORTS                    NAMES
aed84ee21bde  training/webapp:latest python app.py  12 hours ago
  Up 2 seconds 0.0.0.0:49154->5000/tcp  web
```

也可以使用 `docker inspect` 来查看容器的名字

```
$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde
/web
```

注意：容器的名称是唯一的。如果已经命名了一个叫 web 的容器，当你要再次使用 web 这个名称的时候，需要先用 `docker rm` 来删除之前创建的同名容器。

在执行 `docker run` 的时候如果添加 `--rm` 标记，则容器在终止后会立刻删除。注意， `--rm` 和 `-d` 参数不能同时使用。

容器互联

使用 `--link` 参数可以让容器之间安全的进行交互。

下面先创建一个新的数据库容器。

```
$ sudo docker run -d --name db training/postgres
```

删除之前创建的 web 容器

```
$ docker rm -f web
```

然后创建一个新的 web 容器，并将它连接到 db 容器

```
$ sudo docker run -d -P --name web --link db:db training/webapp
python app.py
```

此时，db 容器和 web 容器建立互联关系。

`--link` 参数的格式为 `--link name:alias` ，其中 `name` 是要链接的容器的名称， `alias` 是这个连接的别名。

使用 `docker ps` 来查看容器的连接

```
$ docker ps
CONTAINER ID  IMAGE                     COMMAND               CR
EATED             STATUS              PORTS                 NA
MES
349169744e49  training/postgres:latest  su postgres -c '/usr  Ab
out a minute ago  Up About a minute  5432/tcp              db
, web/db
aed84ee21bde  training/webapp:latest    python app.py         16
 hours ago        Up 2 minutes        0.0.0.0:49154->5000/tcp  we
b
```

可以看到自定义命名的容器，db 和 web，db 容器的 names 列有 db 也有 web/db。这表示 web 容器链接到 db 容器，web 容器将被允许访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主主机上。在启动 db 容器的时候并没有使用 `-p` 和 `-P` 标记，从而避免了暴露数据库端口到外部网络上。

Docker 通过 2 种方式为容器公开连接信息：

- 环境变量
- 更新 `/etc/hosts` 文件

使用 `env` 命令来查看 web 容器的环境变量

```
$ sudo docker run --rm --name web2 --link db:db training/webapp
env
. . .
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5000_TCP=tcp://172.17.0.5:5432
DB_PORT_5000_TCP_PROTO=tcp
DB_PORT_5000_TCP_PORT=5432
DB_PORT_5000_TCP_ADDR=172.17.0.5
. . .
```

其中 DB_ 开头的环境变量是供 web 容器连接 db 容器使用，前缀采用大写的连接别名。

除了环境变量，Docker 还添加 host 信息到父容器的 `/etc/hosts` 的文件。下面是父容器 web 的 hosts 文件

```
$ sudo docker run -t -i --rm --link db:db training/webapp /bin/b
ash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
. . .
172.17.0.5  db
```

这里有 2 个 hosts，第一个是 web 容器，web 容器用 id 作为他的主机名，第二个是 db 容器的 ip 和主机名。 可以在 web 容器中安装 ping 命令来测试跟db容器的连通。

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db
PING db (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

用 ping 来测试db容器，它会解析成 `172.17.0.5` 。*注意：官方的 ubuntu 镜像默认没有安装 ping，需要自行安装。

用户可以链接多个父容器到子容器，比如可以链接多个 web 到 db 容器上。

高级网络配置

本章将介绍 Docker 的一些高级网络配置和选项。

当 Docker 启动时，会自动在主机上创建一个 `docker0` 虚拟网桥，实际上是 Linux 的一个 bridge，可以理解为一个软件交换机。它会在挂载到它的网口之间进行转发。

同时，Docker 随机分配一个本地未占用的私有网段（在 RFC1918 中定义）中的一个地址给 `docker0` 接口。比如典型的 `172.17.42.1` ，掩码为 `255.255.0.0` 。此后启动的容器内的网口也会自动分配一个同一网段（ `172.17.0.0/16` ）的地址。

当创建一个 Docker 容器的时候，同时会创建了一对 `veth pair` 接口（当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包）。这对接口一端在容器内，即 `eth0` ；另一端在本地并被挂载到 `docker0` 网桥，名称以 `veth` 开头（例如 `vethAQI2QT` ）。通过这种方式，主机可以跟容器通信，容器之间也可以相互通信。Docker 就创建了在主机和所有容器之间一个虚拟共享网络。

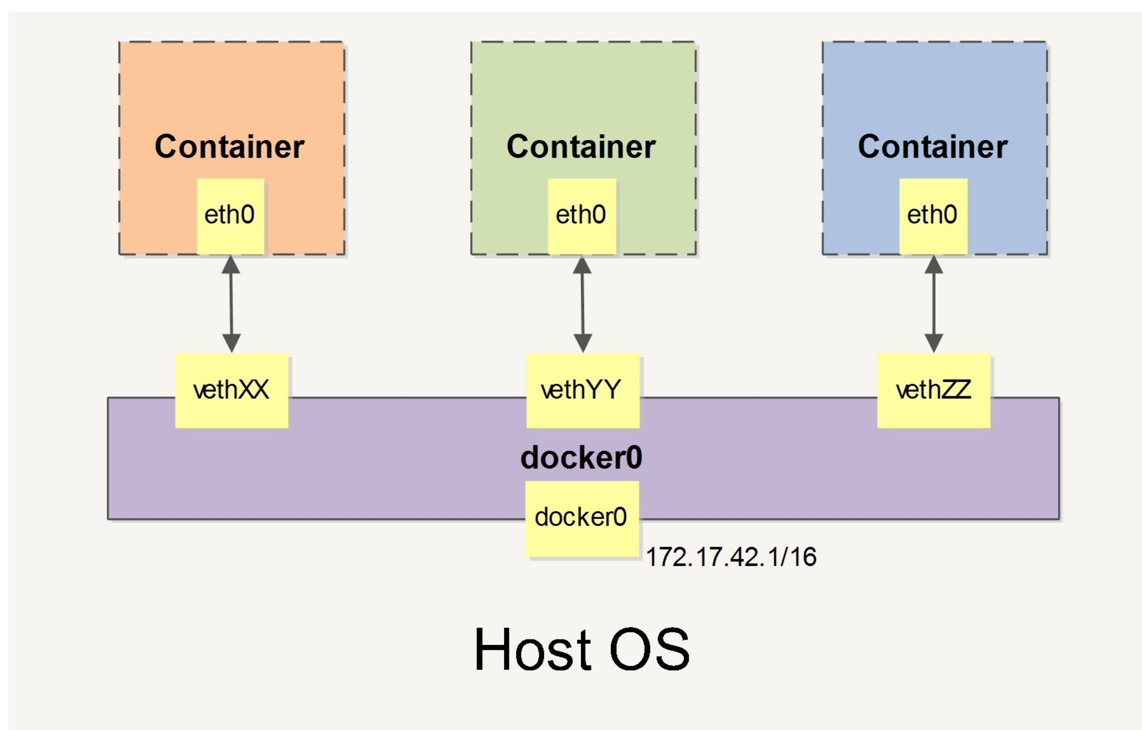


图 *1.12.1 - Docker* 网络

接下来的部分将介绍在一些场景中，Docker 所有的网络定制配置。以及通过 Linux 命令来调整、补充、甚至替换 Docker 默认的网络配置。

快速配置指南

下面是一个跟 Docker 网络相关的命令列表。

其中有些命令选项只有在 Docker 服务启动的时候才能配置，而且不能马上生效。

- `-b BRIDGE or --bridge=BRIDGE` --指定容器挂载的网桥
- `--bip=CIDR` --定制 docker0 的掩码
- `-H SOCKET... or --host=SOCKET...` --Docker 服务端接收命令的通道
- `--icc=true|false` --是否支持容器之间进行通信
- `--ip-forward=true|false` --请看下文容器之间的通信
- `--iptables=true|false` --是否允许 Docker 添加 iptables 规则
- `--mtu=BYTES` --容器网络中的 MTU

下面2个命令选项既可以在启动服务时指定，也可以 Docker 容器启动（ `docker run` ）时候指定。在 Docker 服务启动的时候指定则会成为默认值，后面执行 `docker run` 时可以覆盖设置的默认值。

- `--dns=IP_ADDRESS...` --使用指定的DNS服务器
- `--dns-search=DOMAIN...` --指定DNS搜索域

最后这些选项只有在 `docker run` 执行时使用，因为它是针对容器的特性内容。

- `-h HOSTNAME or --hostname=HOSTNAME` --配置容器主机名
- `--link=CONTAINER_NAME:ALIAS` --添加到另一个容器的连接
- `--net=bridge|none|container:NAME_or_ID|host` --配置容器的桥接模式
- `-p SPEC or --publish=SPEC` --映射容器端口到宿主主机
- `-P or --publish-all=true|false` --映射容器所有端口到宿主主机

配置 DNS

Docker 没有为每个容器专门定制镜像，那么怎么自定义配置容器的主机名和 DNS 配置呢？ 秘诀就是它利用虚拟文件来挂载到来容器的 3 个相关配置文件。

在容器中使用 mount 命令可以看到挂载信息：

```
$ mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
tmpfs on /etc/resolv.conf type tmpfs ...
...
```

这种机制可以让宿主主机 DNS 信息发生更新后，所有 Docker 容器的 dns 配置通过 `/etc/resolv.conf` 文件立刻得到更新。

如果用户想要手动指定容器的配置，可以利用下面的选项。

`-h HOSTNAME or --hostname=HOSTNAME` 设定容器的主机名，它会被写到容器内的 `/etc/hostname` 和 `/etc/hosts` 。但它在容器外部看不到，既不会在 `docker ps` 中显示，也不会在其他的容器的 `/etc/hosts` 看到。

`--link=CONTAINER_NAME:ALIAS` 选项会在创建容器的时候，添加一个其他容器的主机名到 `/etc/hosts` 文件中，让新容器的进程可以使用主机名 ALIAS 就可以连接它。

`--dns=IP_ADDRESS` 添加 DNS 服务器到容器的 `/etc/resolv.conf` 中，让容器用这个服务器来解析所有不在 `/etc/hosts` 中的主机名。

`--dns-search=DOMAIN` 设定容器的搜索域，当设定搜索域为 `.example.com` 时，在搜索一个名为 host 的主机时，DNS 不仅搜索host，还会搜索 `host.example.com` 。 注意：如果没有上述最后 2 个选项，Docker 会默认用主机上的 `/etc/resolv.conf` 来配置容器。

容器访问控制

容器的访问控制，主要通过 Linux 上的 `iptables` 防火墙来进行管理和实现。`iptables` 是 Linux 上默认的防火墙软件，在大部分发行版中都自带。

容器访问外部网络

容器要想访问外部网络，需要本地系统的转发支持。在Linux 系统中，检查转发是否打开。

```
$sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

如果为 0，说明没有开启转发，则需要手动打开。

```
$sysctl -w net.ipv4.ip_forward=1
```

如果在启动 Docker 服务的时候设定 `--ip-forward=true` , Docker 就会自动设定系统的 `ip_forward` 参数为 1。

容器之间访问

容器之间相互访问，需要两方面的支持。

- 容器的网络拓扑是否已经互联。默认情况下，所有容器都会被连接到 `docker0` 网桥上。
- 本地系统的防火墙软件 -- `iptables` 是否允许通过。

访问所有端口

当启动 Docker 服务时候，默认会添加一条转发策略到 iptables 的 FORWARD 链上。策略为通过（ `ACCEPT` ）还是禁止（ `DROP` ）取决于配置 `--icc=true` （缺省值）还是 `--icc=false` 。当然，如果手动指定 `--iptables=false` 则不会添加 `iptables` 规则。

可见，默认情况下，不同容器之间是允许网络互通的。如果为了安全考虑，可以在 `/etc/default/docker` 文件中配置 `DOCKER_OPTS=--icc=false` 来禁止它。

访问指定端口

在通过 `-icc=false` 关闭网络访问后，还可以通过 `--link=CONTAINER_NAME:ALIAS` 选项来访问容器的开放端口。

例如，在启动 Docker 服务时，可以同时使用 `icc=false --iptables=true` 参数来关闭允许相互的网络访问，并让 Docker 可以修改系统中的 `iptables` 规则。

此时，系统中的 `iptables` 规则可能是类似

```
$ sudo iptables -nL
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  0.0.0.0/0            0.0.0.0/0
...
```

之后，启动容器（`docker run`）时使用 `--link=CONTAINER_NAME:ALIAS` 选项。Docker 会在 `iptable` 中为 两个容器分别添加一条 `ACCEPT` 规则，允许相互访问开放的端口（取决于 Dockerfile 中的 EXPOSE 行）。

当添加了 `--link=CONTAINER_NAME:ALIAS` 选项后，添加了 `iptables` 规则。

```
$ sudo iptables -nL
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  172.17.0.2           172.17.0.3           tc
p spt:80
ACCEPT     tcp  --  172.17.0.3           172.17.0.2           tc
p dpt:80
DROP       all  --  0.0.0.0/0            0.0.0.0/0
```

注意： `--link=CONTAINER_NAME:ALIAS` 中的 `CONTAINER_NAME` 目前必须是 Docker 分配的名字，或使用 `--name` 参数指定的名字。主机名则不会被识别。

映射容器端口到宿主主机的实现

默认情况下，容器可以主动访问到外部网络的连接，但是外部网络无法访问到容器。

容器访问外部实现

容器所有到外部网络的连接，源地址都会被NAT成本地系统的IP地址。这是使用 `iptables` 的源地址伪装操作实现的。

查看主机的 NAT 规则。

```
$ sudo iptables -t nat -nL
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16       !172.17.0.0/16
...
```

其中，上述规则将所有源地址在 `172.17.0.0/16` 网段，目标地址为其他网段（外部网络）的流量动态伪装为从系统网卡发出。MASQUERADE 跟传统 SNAT 的好处是它能动态从网卡获取地址。

外部访问容器实现

容器允许外部访问，可以在 `docker run` 时候通过 `-p` 或 `-P` 参数来启用。

不管用那种办法，其实也是在本地的 `iptable` 的 nat 表中添加相应的规则。

使用 `-P` 时：

```
$ iptables -t nat -nL
...
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tc
p dpt:49153 to:172.17.0.2:80
```

使用 `-p 80:80` 时：

```
$ iptables -t nat -nL
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tc
p dpt:80 to:172.17.0.2:80
```

注意：

- 这里的规则映射了 0.0.0.0，意味着将接受主机来自所有接口的流量。用户可以通过 `-p IP:host_port:container_port` 或 `-p IP::port` 来指定允许访问容器的主机上的 IP、接口等，以制定更严格的规则。
- 如果希望永久绑定到某个固定的 IP 地址，可以在 Docker 配置文件 `/etc/default/docker` 中指定 `DOCKER_OPTS="--ip=IP_ADDRESS"`，之后重启 Docker 服务即可生效。

配置 docker0 网桥

Docker 服务默认会创建一个 `docker0` 网桥（其上有一个 `docker0` 内部接口），它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和本地主机都放到同一个物理网络。

Docker 默认指定了 `docker0` 接口 的 IP 地址和子网掩码，让主机和容器之间可以通过网桥相互通信，它还给出了 MTU（接口允许接收的最大传输单元），通常是 1500 Bytes，或宿主主机网络路由上支持的默认值。这些值都可以在服务启动的时候进行配置。

- `--bip=CIDR` -- IP 地址加掩码格式，例如 192.168.1.5/24
- `--mtu=BYTES` -- 覆盖默认的 Docker mtu 配置

也可以在配置文件中配置 DOCKER_OPTS，然后重启服务。 由于目前 Docker 网桥是 Linux 网桥，用户可以使用 `brctl show` 来查看网桥和端口连接信息。

```
$ sudo brctl show
bridge name     bridge id               STP enabled     interfac
es
docker0         8000.3a1d7362b4ee       no              veth65f9
                                                vethdda6
```

*注： `brctl` 命令在 Debian、Ubuntu 中可以使用 `sudo apt-get install bridge-utils` 来安装。

每次创建一个新容器的时候，Docker 从可用的地址段中选择一个空闲的 IP 地址分配给容器的 eth0 端口。使用本地主机上 `docker0` 接口的 IP 作为所有容器的默认网关。

```
$ sudo docker run -i -t --rm base /bin/bash
$ ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast stat
e UP group default qlen 1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
       valid_lft forever preferred_lft forever
$ ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3
$ exit
```

自定义网桥

除了默认的 `docker0` 网桥，用户也可以指定网桥来连接各个容器。

在启动 Docker 服务的时候，使用 `-b BRIDGE` 或 `--bridge=BRIDGE` 来指定使用的网桥。

如果服务已经运行，那需要先停止服务，并删除旧的网桥。

```
$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
```

然后创建一个网桥 `bridge0` 。

```
$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.5.1/24 dev bridge0
$ sudo ip link set dev bridge0 up
```

查看确认网桥创建并启动。

```
$ ip addr show bridge0
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP g
roup default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
```

配置 Docker 服务，默认桥接到创建的网桥上。

```
$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker start
```

启动 Docker 服务。 新建一个容器，可以看到它已经桥接到了 `bridge0` 上。

可以继续用 `brctl show` 命令查看桥接的信息。另外，在容器中可以使用 `ip addr` 和 `ip route` 命令来查看 IP 地址配置和路由信息。

工具和示例

在介绍自定义网络拓扑之前，你可能会对一些外部工具和例子感兴趣：

pipework

Jérôme Petazzoni 编写了一个叫 pipework 的 shell 脚本，可以帮助用户在比较复杂的场景中完成容器的连接。

playground

Brandon Rhodes 创建了一个提供完整的 Docker 容器网络拓扑管理的 Python库，包括路由、NAT 防火墙；以及一些提供 HTTP, SMTP, POP, IMAP, Telnet, SSH, FTP 的服务器。

编辑网络配置文件

Docker 1.2.0 开始支持在运行中的容器里编辑 `/etc/hosts` , `/etc/hostname` 和 `/etc/resolve.conf` 文件。

但是这些修改是临时的，只在运行的容器中保留，容器终止或重启后并不会被保存下来。也不会被 `docker commit` 提交。

示例：创建一个点到点连接

默认情况下，Docker 会将所有容器连接到由 `docker0` 提供的虚拟子网中。

用户有时候需要两个容器之间可以直连通信，而不用通过主机网桥进行桥接。

解决办法很简单：创建一对 `peer` 接口，分别放到两个容器中，配置成点到点链路类型即可。

首先启动 2 个容器：

```
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@1f1f4c1f931a:/#
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@12e343489d2f:/#
```

找到进程号，然后创建网络命名空间的跟踪文件。

```
$ sudo docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a
2989
$ sudo docker inspect -f '{{.State.Pid}}' 12e343489d2f
3004
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/2989/ns/net /var/run/netns/2989
$ sudo ln -s /proc/3004/ns/net /var/run/netns/3004
```

创建一对 `peer` 接口，然后配置路由

```
$ sudo ip link add A type veth peer name B

$ sudo ip link set A netns 2989
$ sudo ip netns exec 2989 ip addr add 10.1.1.1/32 dev A
$ sudo ip netns exec 2989 ip link set A up
$ sudo ip netns exec 2989 ip route add 10.1.1.2/32 dev A

$ sudo ip link set B netns 3004
$ sudo ip netns exec 3004 ip addr add 10.1.1.2/32 dev B
$ sudo ip netns exec 3004 ip link set B up
$ sudo ip netns exec 3004 ip route add 10.1.1.1/32 dev B
```

现在这 2 个容器就可以相互 ping 通，并成功建立连接。点到点链路不需要子网和子网掩码。

此外，也可以不指定 `--net=none` 来创建点到点链路。这样容器还可以通过原先的网络来通信。

利用类似的办法，可以创建一个只跟主机通信的容器。但是一般情况下，更推荐使用 `--icc=false` 来关闭容器之间的通信。

安全

评估 Docker 的安全性时，主要考虑三个方面:

- 由内核的命名空间和控制组机制提供的容器内在安全
- Docker 程序（特别是服务端）本身的抗攻击性
- 内核安全性的加强机制对容器安全性的影响

内核命名空间

Docker 容器和 LXC 容器很相似，所提供的安全特性也差不多。当用 `docker run` 启动一个容器时，在后台 Docker 为容器创建了一个独立的命名空间和控制组集合。

命名空间提供了最基础也是最直接的隔离，在容器中运行的进程不会被运行在主机上的进程和其它容器发现和作用。

每个容器都有自己独有的网络栈，意味着它们不能访问其他容器的 sockets 或接口。不过，如果主机系统上做了相应的设置，容器可以像跟主机交互一样的和其他容器交互。当指定公共端口或使用 links 来连接 2 个容器时，容器就可以相互通信了（可以根据配置来限制通信的策略）。

从网络架构的角度来看，所有的容器通过本地主机的网桥接口相互通信，就像物理机器通过物理交换机通信一样。

那么，内核中实现命名空间和私有网络的代码是否足够成熟？

内核命名空间从 2.6.15 版本（2008 年 7 月发布）之后被引入，数年间，这些机制的可靠性在诸多大型生产系统中被实践验证。

实际上，命名空间的想法和设计提出的时间要更早，最初是为了在内核中引入一种机制来实现 OpenVZ 的特性。 而 OpenVZ 项目早在 2005 年就发布了，其设计和实现都已经十分成熟。

控制组

控制组是 Linux 容器机制的另外一个关键组件，负责实现资源的审计和限制。

它提供了很多有用的特性；以及确保各个容器可以公平地分享主机的内存、CPU、磁盘 IO 等资源；当然，更重要的是，控制组确保了当容器内的资源使用产生压力时不会连累主机系统。

尽管控制组不负责隔离容器之间相互访问、处理数据和进程，它在防止拒绝服务（DDOS）攻击方面是必不可少的。尤其是在多用户的平台（比如公有或私有的 PaaS）上，控制组十分重要。例如，当某些应用程序表现异常的时候，可以保证一致地正常运行和性能。

控制组机制始于 2006 年，内核从 2.6.24 版本开始被引入。

Docker服务端的防护

运行一个容器或应用程序的核心是通过 Docker 服务端。Docker 服务的运行目前需要 root 权限，因此其安全性十分关键。

首先，确保只有可信的用户才可以访问 Docker 服务。Docker 允许用户在主机和容器间共享文件夹，同时不需要限制容器的访问权限，这就容易让容器突破资源限制。例如，恶意用户启动容器的时候将主机的根目录 `/` 映射到容器的 `/host` 目录中，那么容器理论上就可以对主机的文件系统进行任意修改了。这听起来很疯狂？但是事实上几乎所有虚拟化系统都允许类似的资源共享，而没法禁止用户共享主机根文件系统到虚拟机系统。

这将会造成很严重的安全后果。因此，当提供容器创建服务时（例如通过一个 web 服务器），要更加注意进行参数的安全检查，防止恶意的用户用特定参数来创建一些破坏性的容器。

为了加强对服务端的保护，Docker 的 REST API（客户端用来跟服务端通信）在 0.5.2 之后使用本地的 Unix 套接字机制替代了原先绑定在 127.0.0.1 上的 TCP 套接字，因为后者容易遭受跨站脚本攻击。现在用户使用 Unix 权限检查来加强套接字的访问安全。

用户仍可以利用 HTTP 提供 REST API 访问。建议使用安全机制，确保只有可信的网络或 VPN，或证书保护机制（例如受保护的 stunnel 和 ssl 认证）下的访问可以进行。此外，还可以使用 HTTPS 和证书来加强保护。

最近改进的 Linux 命名空间机制将可以实现使用非 root 用户来运行全功能的容器。这将从根本上解决了容器和主机之间共享文件系统而引起的安全问题。

终极目标是改进 2 个重要的安全特性：

- 将容器的 root 用户映射到本地主机上的非 root 用户，减轻容器和主机之间因权限提升而引起的安全问题；
- 允许 Docker 服务端在非 root 权限下运行，利用安全可靠的子进程来代理执行需要特权权限的操作。这些子进程将只允许在限定范围内进行操作，例如仅仅负责虚拟网络设定或文件系统管理、配置操作等。

最后，建议采用专用的服务器来运行 Docker 和相关的管理服务（例如管理服务比如 ssh 监控和进程监控、管理工具 nrpe、collectd 等）。其它的业务服务都放到容器中去运行。

内核能力机制

能力机制（Capability）是 Linux 内核一个强大的特性，可以提供细粒度的权限访问控制。 Linux 内核自 2.2 版本起就支持能力机制，它将权限划分为更加细粒度的操作能力，既可以作用在进程上，也可以作用在文件上。

例如，一个 Web 服务进程只需要绑定一个低于 1024 的端口的权限，并不需要 root 权限。那么它只需要被授权 `net_bind_service` 能力即可。此外，还有很多其他的类似能力来避免进程获取 root 权限。

默认情况下，Docker 启动的容器被严格限制只允许使用内核的一部分能力。

使用能力机制对加强 Docker 容器的安全有很多好处。通常，在服务器上会运行一堆需要特权权限的进程，包括有 ssh、cron、syslogd、硬件管理工具模块（例如负载模块）、网络配置工具等等。容器跟这些进程是不同的，因为几乎所有的特权进程都由容器以外的支持系统来进行管理。

- ssh 访问被主机上ssh服务来管理；
- cron 通常应该作为用户进程执行，权限交给使用它服务的应用来处理；
- 日志系统可由 Docker 或第三方服务管理；
- 硬件管理无关紧要，容器中也就无需执行 udevd 以及类似服务；
- 网络管理也都在主机上设置，除非特殊需求，容器不需要对网络进行配置。

从上面的例子可以看出，大部分情况下，容器并不需要“真正的” root 权限，容器只需要少数的能力即可。为了加强安全，容器可以禁用一些没必要的权限。

- 完全禁止任何 mount 操作；
- 禁止直接访问本地主机的套接字；
- 禁止访问一些文件系统的操作，比如创建新的设备、修改文件属性等；
- 禁止模块加载。

这样，就算攻击者在容器中取得了 root 权限，也不能获得本地主机的较高权限，能进行的破坏也有限。

默认情况下，Docker采用白名单机制，禁用必需功能之外的其它权限。 当然，用户也可以根据自身需求来为 Docker 容器启用额外的权限。

其它安全特性

除了能力机制之外，还可以利用一些现有的安全机制来增强使用 Docker 的安全性，例如 TOMOYO, AppArmor, SELinux, GRSEC 等。

Docker 当前默认只启用了能力机制。用户可以采用多种方案来加强 Docker 主机的安全，例如：

- 在内核中启用 GRSEC 和 PAX，这将增加很多编译和运行时的安全检查；通过地址随机化避免恶意探测等。并且，启用该特性不需要 Docker 进行任何配置。
- 使用一些有增强安全特性的容器模板，比如带 AppArmor 的模板和 Redhat 带 SELinux 策略的模板。这些模板提供了额外的安全特性。
- 用户可以自定义访问控制机制来定制安全策略。

跟其它添加到 Docker 容器的第三方工具一样（比如网络拓扑和文件系统共享），有很多类似的机制，在不改变 Docker 内核情况下就可以加固现有的容器。

总结

总体来看，Docker 容器还是十分安全的，特别是在容器内不使用 root 权限来运行进程的话。

另外，用户可以使用现有工具，比如 Apparmor, SELinux, GRSEC 来增强安全性；甚至自己在内核中实现更复杂的安全机制。

底层实现

Docker 底层的核心技术包括 Linux 上的命名空间（Namespaces）、控制组（Control groups）、Union 文件系统（Union file systems）和容器格式（Container format）。

我们知道，传统的虚拟机通过在宿主主机中运行 hypervisor 来模拟一整套完整的硬件环境提供给虚拟机的操作系统。虚拟机系统看到的环境是可限制的，也是彼此隔离的。 这种直接的做法实现了对资源最完整的封装，但很多时候往往意味着系统资源的浪费。 例如，以宿主机和虚拟机系统都为 Linux 系统为例，虚拟机中运行的应用其实可以利用宿主机系统中的运行环境。

我们知道，在操作系统中，包括内核、文件系统、网络、PID、UID、IPC、内存、硬盘、CPU 等等，所有的资源都是应用进程直接共享的。 要想实现虚拟化，除了要实现对内存、CPU、网络IO、硬盘IO、存储空间等的限制外，还要实现文件系统、网络、PID、UID、IPC等等的相互隔离。 前者相对容易实现一些，后者则需要宿主机系统的深入支持。

随着 Linux 系统对于命名空间功能的完善实现，程序员已经可以实现上面的所有需求，让某些进程在彼此隔离的命名空间中运行。大家虽然都共用一个内核和某些运行时环境（例如一些系统命令和系统库），但是彼此却看不到，都以为系统中只有自己的存在。这种机制就是容器（Container），利用命名空间来做权限的隔离控制，利用 cgroups 来做资源分配。

基本架构

Docker 采用了 C/S架构，包括客户端和服务端。 Docker daemon 作为服务端接受来自客户户的请求，并处理这些请求（创建、运行、分发容器）。 客户端和服务端既可以运行在一个机器上，也可通过 socket 或者 RESTful API 来进行通信。

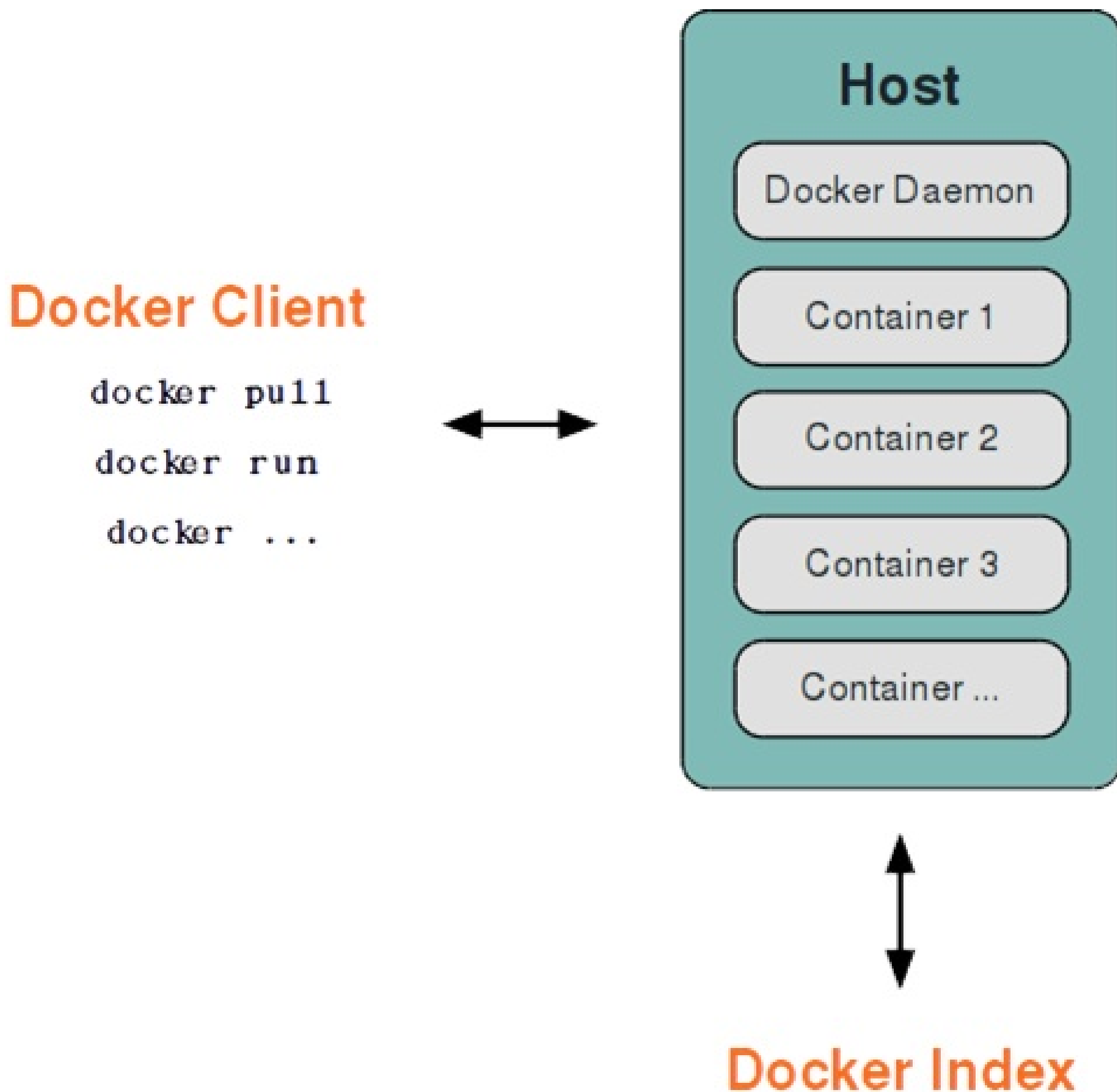


图 *1.14.1.1 - Docker* 基本架构

Docker daemon 一般在宿主主机后台运行，等待接收来自客户端的消息。 Docker 客户端则为用户提供一系列可执行命令，用户用这些命令实现跟 Docker daemon 交互。

命名空间

命名空间是 Linux 内核一个强大的特性。每个容器都有自己单独的命名空间，运行在其中的应用都像是在独立的操作系统中运行一样。命名空间保证了容器之间彼此互不影响。

pid 命名空间

不同用户的进程就是通过 pid 命名空间隔离开的，且不同命名空间中可以有相同 pid。所有的 LXC 进程在 Docker 中的父进程为Docker进程，每个 LXC 进程具有不同的命名空间。同时由于允许嵌套，因此可以很方便的实现嵌套的 Docker 容器。

net 命名空间

有了 pid 命名空间, 每个命名空间中的 pid 能够相互隔离，但是网络端口还是共享 host 的端口。网络隔离是通过 net 命名空间实现的， 每个 net 命名空间有独立的网络设备, IP 地址, 路由表, /proc/net 目录。这样每个容器的网络就能隔离开来。Docker 默认采用 veth 的方式，将容器中的虚拟网卡同 host 上的一 个Docker 网桥 docker0 连接在一起。

ipc 命名空间

容器中进程交互还是采用了 Linux 常见的进程间交互方法(interprocess communication - IPC), 包括信号量、消息队列和共享内存等。然而同 VM 不同的是，容器的进程间交互实际上还是 host 上具有相同 pid 命名空间中的进程间交互，因此需要在 IPC 资源申请时加入命名空间信息，每个 IPC 资源有一个唯一的 32 位 id。

mnt 命名空间

类似 chroot，将一个进程放到一个特定的目录执行。mnt 命名空间允许不同命名空间的进程看到的文件结构不同，这样每个命名空间 中的进程所看到的文件目录就被隔离开了。同 chroot 不同，每个命名空间中的容器在 /proc/mounts 的信息只包含所在命名空间的 mount point。

uts 命名空间

UTS("UNIX Time-sharing System") 命名空间允许每个容器拥有独立的 hostname 和 domain name, 使其在网络上可以被视作一个独立的节点而非 主机上的一个进程。

user 命名空间

每个容器可以有不同的用户和组 id, 也就是说可以在容器内用容器内部的用户执行程序而非主机上的用户 。

*注：关于 Linux 上的命名空间，这篇文章 介绍的很好。

控制组

控制组（cgroups）是 Linux 内核的一个特性，主要用来对共享资源进行隔离、限制、审计等。只有能控制分配到容器的资源，才能避免当多个容器同时运行时的对系统资源的竞争。

控制组技术最早是由 Google 的程序员 2006 年起提出，Linux 内核自 2.6.24 开始支持。

控制组可以提供对容器的内存、CPU、磁盘 IO 等资源的限制和审计管理。

联合文件系统

联合文件系统（UnionFS）是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)。

联合文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。

另外，不同 Docker 容器就可以共享一些基础的文件系统层，同时再加上自己独有的改动层，大大提高了存储的效率。

Docker 中使用的 AUFS（AnotherUnionFS）就是一种联合文件系统。 AUFS 支持为每一个成员目录（类似 Git 的分支）设定只读（readonly）、读写（readwrite）和写出（whiteout-able）权限, 同时 AUFS 里有一个类似分层的概念, 对只读权限的分支可以逻辑上进行增量地修改(不影响只读部分的)。

Docker 目前支持的联合文件系统种类包括 AUFS, btrfs, vfs 和 DeviceMapper。

容器格式

最初，Docker 采用了 LXC 中的容器格式。自 1.20 版本开始，Docker 也开始支持新的 libcontainer 格式，并作为默认选项。

对更多容器格式的支持，还在进一步的发展中。

Docker 网络实现

Docker 的网络实现其实就是利用了 Linux 上的网络命名空间和虚拟网络设备（特别是 veth pair）。建议先熟悉了解这两部分的基本概念再阅读本章。

基本原理

首先，要实现网络通信，机器需要至少一个网络接口（物理接口或虚拟接口）来收发数据包；此外，如果不同子网之间要进行通信，需要路由机制。

Docker 中的网络接口默认都是虚拟的接口。虚拟接口的优势之一是转发效率较高。 Linux 通过在内核中进行数据复制来实现虚拟接口之间的数据转发，发送接口的发送缓存中的数据包被直接复制到接收接口的接收缓存中。对于本地系统和容器内系统看来就像是一个正常的以太网卡，只是它不需要真正同外部网络设备通信，速度要快很多。

Docker 容器网络就利用了这项技术。它在本地主机和容器内分别创建一个虚拟接口，并让它们彼此连通（这样的一对接口叫做 `veth pair` ）。

创建网络参数

Docker 创建一个容器的时候，会执行如下操作：

- 创建一对虚拟接口，分别放到本地主机和新容器中；
- 本地主机一端桥接到默认的 docker0 或指定网桥上，并具有一个唯一的名字，如 veth65f9；
- 容器一端放到新容器中，并修改名字作为 eth0，这个接口只在容器的命名空间可见；
- 从网桥可用地址段中获取一个空闲地址分配给容器的 eth0，并配置默认路由到桥接网卡 veth65f9。

完成这些之后，容器就可以使用 eth0 虚拟网卡来连接其他容器和其他网络。

可以在 `docker run` 的时候通过 `--net` 参数来指定容器的网络配置，有4个可选值：

- `--net=bridge` 这个是默认值，连接到默认的网桥。

- `--net=host` 告诉 Docker 不要将容器网络放到隔离的命名空间中，即不要容器化容器内的网络。此时容器使用本地主机的网络，它拥有完全的本地主机接口访问权限。容器进程可以跟主机其它 root 进程一样可以打开低范围的端口，可以访问本地网络服务比如 D-bus，还可以让容器做一些影响整个主机系统的事情，比如重启主机。因此使用这个选项的时候要非常小心。如果进一步的使用 `--privileged=true`，容器会被允许直接配置主机的网络堆栈。
- `--net=container:NAME_or_ID` 让 Docker 将新建容器的进程放到一个已存在容器的网络栈中，新容器进程有自己的文件系统、进程列表和资源限制，但会和已存在的容器共享 IP 地址和端口等网络资源，两者进程可以直接通过 `lo` 环回接口通信。
- `--net=none` 让 Docker 将新容器放到隔离的网络栈中，但是不进行网络配置。之后，用户可以自己进行配置。

网络配置细节

用户使用 `--net=none` 后，可以自行配置网络，让容器达到跟平常一样具有访问网络的权限。通过这个过程，可以了解 Docker 配置网络的细节。

首先，启动一个 `/bin/bash` 容器，指定 `--net=none` 参数。

```
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@63f36fc01b5f:/#
```

在本地主机查找容器的进程 id，并为它创建网络命名空间。

```
$ sudo docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid
```

检查桥接网卡的 IP 和子网掩码信息。

```
$ ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...
```

创建一对 “veth pair” 接口 A 和 B，绑定 A 到网桥 `docker0`，并启用它

```
$ sudo ip link add A type veth peer name B
$ sudo brctl addif docker0 A
$ sudo ip link set A up
```

将B放到容器的网络命名空间，命名为 eth0，启动它并配置一个可用 IP（桥接网段）和默认网关。

```
$ sudo ip link set B netns $pid
$ sudo ip netns exec $pid ip link set dev B name eth0
$ sudo ip netns exec $pid ip link set eth0 up
$ sudo ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0
$ sudo ip netns exec $pid ip route add default via 172.17.42.1
```

以上，就是 Docker 配置网络的具体过程。

当容器结束后，Docker 会清空容器，容器内的 eth0 会随网络命名空间一起被清除，A 接口也被自动从 `docker0` 卸载。

此外，用户可以使用 `ip netns exec` 命令来在指定网络命名空间中进行配置，从而配置容器内的网络。

Docker Compose 项目

Docker Compose 是 Docker 官方编排（Orchestration）项目之一，负责快速在集群中部署分布式应用。

本章将介绍 Compose 项目情况以及安装和使用。

Compose 简介

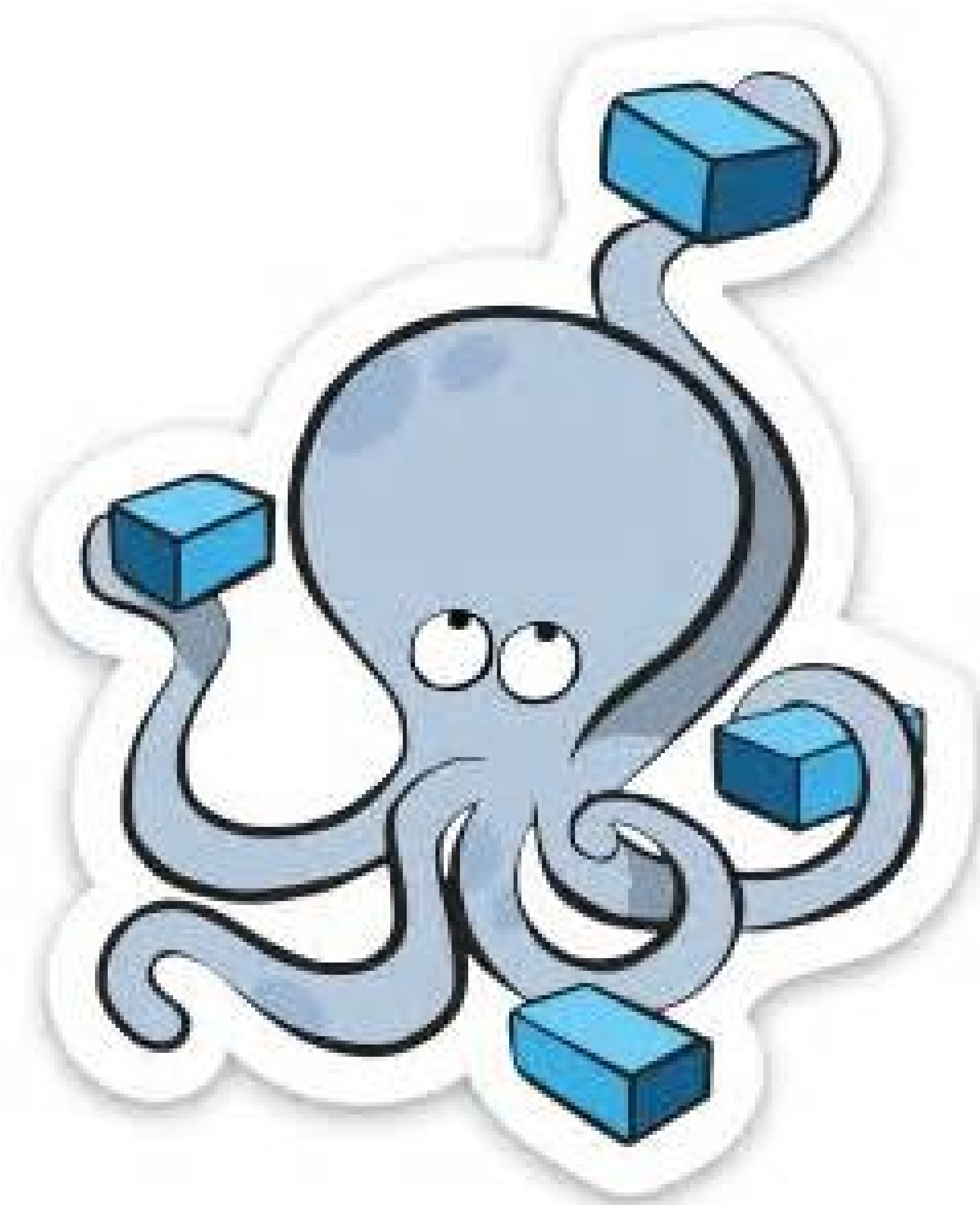


图 *1.15.1.1 - Docker Compose* 项目

Compose 项目是 Docker 官方的开源项目，负责实现对 Docker 容器集群的快速编排。从功能上看，跟 OpenStack 中的 Heat 十分类似。

其代码目前在 https://github.com/docker/compose 上开源。

Compose 定位是“定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker applications）”，其前身是开源项目 Fig，目前仍然兼容 Fig 格式的模板文件。

通过第一部分中的介绍，我们知道使用一个 Dockerfile 模板文件，可以让用户很方便的定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等。

Compose 恰好满足了这样的需求。它允许用户通过一个单独的 `docker-compose.yml` 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

Compose 中有两个重要的概念：

- 服务（service）：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
- 项目(project)：由一组关联的应用容器组成的一个完整业务单元，在 `docker-compose.yml` 文件中定义。

Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。

Compose 项目由 Python 编写，实现上调用了 Docker 服务提供的 API 来对容器进行管理。因此，只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

安装与卸载

Compose 目前支持 Linux 和 Mac OS 平台，两者的安装过程大同小异。

安装 Compose 之前，要先安装 Docker（需要 Docker Engine 1.7.1+），请参考第一部分中章节，在此不再赘述。

Compose 可以通过 Python 的 pip 工具进行安装，可以直接下载编译好的二进制文件使用，甚至直接运行在 Docker 容器中。

前两种方式是传统方式，适合本地环境下安装使用；最后一种方式则不破坏系统环境，更适合云计算场景。

PIP 安装

这种方式是将 Compose 当作一个 Python 应用来从 pip 源中安装。

执行安装命令：

```
$ sudo pip install -U docker-compose
```

可以看到类似如下输出，说明安装成功。

```
Collecting docker-compose
  Downloading docker-compose-1.8.0.tar.gz (149kB): 149kB downloa
ded
...
Successfully installed docker-compose cached-property requests t
exttable websocket-client docker-py dockerpty six enum34 backpor
ts.ssl-match-hostname ipaddress
```

安装成功后，可以查看 `docker-compose` 命令的用法。

```
$ docker-compose -h
Define and run multi-container applications with Docker.

Usage:
```

```
  docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
  docker-compose -h|--help

Options:
  -f, --file FILE           Specify an alternate compose file (d
efault: docker-compose.yml)
  -p, --project-name NAME   Specify an alternate project name (d
efault: directory name)
  --x-networking            (EXPERIMENTAL) Use new Docker networ
king functionality.
                            Requires Docker 1.9 or later.
  --x-network-driver DRIVER (EXPERIMENTAL) Specify a network dri
ver (default: "bridge").
                            Requires Docker 1.9 or later.
  --verbose                 Show more output
  -v, --version             Print version and exit

Commands:
  build              Build or rebuild services
  help               Get help on a command
  kill               Kill containers
  logs               View output from containers
  pause              Pause services
  port               Print the public port for a port binding
  ps                 List containers
  pull               Pulls service images
  restart            Restart services
  rm                 Remove stopped containers
  run                Run a one-off command
  scale              Set number of containers for a service
  start              Start services
  stop               Stop services
  unpause            Unpause services
  up                 Create and start containers
  migrate-to-labels  Recreate containers to add labels
  version            Show the Docker-Compose version information
```

之后，可以添加 bash 补全命令。

```
$ curl -L https://raw.githubusercontent.com/docker/compose/1.8.0
/contrib/completion/bash/docker-compose > /etc/bash_completion.d
/docker-compose
```

二进制包

官方定义编译好二进制包，供大家使用。这些发布的二进制包可以在 https://github.com/docker/compose/releases 页面找到。

这些二进制文件，下载后直接放到执行路径下，并添加执行权限即可。

例如，在 Linux 平台上。

```
$ sudo curl -L https://github.com/docker/compose/releases/downlo
ad/1.8.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/d
ocker-compose
$ sudo chmod a+x /usr/local/bin/docker-compose
```

可以使用 `docker-compose version` 命令来查看版本信息，以测试是否安装成功。

```
$ docker-compose version
docker-compose version 1.8.0, build 94f7016
docker-py version: 1.9.0
CPython version: 2.7.6
OpenSSL version: OpenSSL 1.0.1f 6 Jan 2014
```

容器中执行

Compose 既然是一个 Python 应用，自然也可以直接用容器来执行它。

```
$ curl -L https://github.com/docker/compose/releases/download/1.
8.0/run.sh > /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
```

实际上，查看下载的 `run.sh` 脚本内容，如下

```
set -e

VERSION="1.8.0"
IMAGE="docker/compose:$VERSION"


# Setup options for connecting to docker host
if [ -z "$DOCKER_HOST" ]; then
    DOCKER_HOST="/var/run/docker.sock"
fi
if [ -S "$DOCKER_HOST" ]; then
    DOCKER_ADDR="-v $DOCKER_HOST:$DOCKER_HOST -e DOCKER_HOST"
else
    DOCKER_ADDR="-e DOCKER_HOST -e DOCKER_TLS_VERIFY -e DOCKER_C
ERT_PATH"
fi


# Setup volume mounts for compose config and context
if [ "$(pwd)" != '/' ]; then
    VOLUMES="-v $(pwd):$(pwd)"
fi
if [ -n "$COMPOSE_FILE" ]; then
    compose_dir=$(dirname $COMPOSE_FILE)
fi
# TODO: also check --file argument
if [ -n "$compose_dir" ]; then
    VOLUMES="$VOLUMES -v $compose_dir:$compose_dir"
fi
if [ -n "$HOME" ]; then
    VOLUMES="$VOLUMES -v $HOME:$HOME -v $HOME:/root" # mount $HO
ME in /root to share docker.config
fi

# Only allocate tty if we detect one
if [ -t 1 ]; then
    DOCKER_RUN_OPTIONS="-t"
fi
if [ -t 0 ]; then
```

```
    DOCKER_RUN_OPTIONS="$DOCKER_RUN_OPTIONS -i"
fi

exec docker run --rm $DOCKER_RUN_OPTIONS $DOCKER_ADDR $COMPOSE_O
PTIONS $VOLUMES -w "$(pwd)" $IMAGE "$@"
```

可以看到，它其实是下载了 `docker/compose` 镜像并运行。

卸载

如果是二进制包方式安装的，删除二进制文件即可。

```
$ sudo rm /usr/local/bin/docker-compose
```

如果是通过 python pip 工具安装的，则可以执行如下命令删除。

```
$ sudo pip uninstall docker-compose
```

使用

术语

首先介绍几个术语。

- 服务（service）：一个应用容器，实际上可以运行多个相同镜像的实例。
- 项目(project)：由一组关联的应用容器组成的一个完整业务单元。

可见，一个项目可以由多个服务（容器）关联而成，Compose 面向项目进行管理。

场景

下面，我们创建一个经典的 Web 项目：一个 Haproxy，挂载三个 Web 容器。

创建一个 `compose-haproxy-web` 目录，作为项目工作目录，并在其中分别创建两个子目录： `haproxy` 和 `web` 。

web 子目录

这里用 Python 程序来提供一个简单的 HTTP 服务，打印出访问者的 IP 和 实际的本地 IP。

index.py

编写一个 `index.py` 作为服务器文件，代码为

```
#!/usr/bin/python
#authors: yeasy.github.com
#date: 2013-07-05

import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler
import socket
import fcntl
```

```
import struct
import pickle
from datetime import datetime
from collections import OrderedDict

class HandlerClass(SimpleHTTPRequestHandler):
    def get_ip_address(self,ifname):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        return socket.inet_ntoa(fcntl.ioctl(
            s.fileno(),
            0x8915,  # SIOCGIFADDR
            struct.pack('256s', ifname[:15])
        )[20:24])
    def log_message(self, format, *args):
        if len(args) < 3 or "200" not in args[1]:
            return
        try:
            request = pickle.load(open("pickle_data.txt","r"))
        except:
            request=OrderedDict()
        time_now = datetime.now()
        ts = time_now.strftime('%Y-%m-%d %H:%M:%S')
        server = self.get_ip_address('eth0')
        host=self.address_string()
        addr_pair = (host,server)
        if addr_pair not in request:
            request[addr_pair]=[1,ts]
        else:
            num = request[addr_pair][0]+1
            del request[addr_pair]
            request[addr_pair]=[num,ts]
        file=open("index.html", "w")
        file.write("<!DOCTYPE html> <html> <body><center><h1><fo
nt color=\"blue\" face=\"Georgia, Arial\" size=8><em>HA</em></fo
nt> Webpage Visit Results</h1></center>")
        for pair in request:
            if pair[0] == host:
                guest = "LOCAL: "+pair[0]
            else:
                guest = pair[0]
```

```
            if (time_now-datetime.strptime(request[pair][1],'%Y-
%m-%d %H:%M:%S')).seconds < 3:
                file.write("<p style=\"font-size:150%\" >#"+ str
(request[pair][1]) +": <font color=\"red\">"+str(request[pair][0
])+ "</font> requests " + "from &lt<font color=\"blue\">"+guest+
"</font>&gt to WebServer &lt<font color=\"blue\">"+pair[1]+"</fo
nt>&gt</p>")
            else:
                file.write("<p style=\"font-size:150%\" >#"+ str
(request[pair][1]) +": <font color=\"maroon\">"+str(request[pair
][0])+ "</font> requests " + "from &lt<font color=\"navy\">"+gue
st+"</font>&gt to WebServer &lt<font color=\"navy\">"+pair[1]+"<
/font>&gt</p>")
        file.write("</body> </html>")
        file.close()
        pickle.dump(request,open("pickle_data.txt","w"))

if __name__ == '__main__':
    try:
        ServerClass  = BaseHTTPServer.HTTPServer
        Protocol     = "HTTP/1.0"
        addr = len(sys.argv) < 2 and "0.0.0.0" or sys.argv[1]
        port = len(sys.argv) < 3 and 80 or int(sys.argv[2])
        HandlerClass.protocol_version = Protocol
        httpd = ServerClass((addr, port), HandlerClass)
        sa = httpd.socket.getsockname()
        print "Serving HTTP on", sa[0], "port", sa[1], "..."
        httpd.serve_forever()
    except:
        exit()
```

index.html

生成一个临时的 `index.html` 文件，其内容会被 index.py 更新。

```
$ touch index.html
```

Dockerfile

生成一个 Dockerfile，内容为

```
FROM python:2.7
WORKDIR /code
ADD . /code
EXPOSE 80
CMD python index.py
```

haproxy 目录

在其中生成一个 `haproxy.cfg` 文件，内容为

```
global
  log 127.0.0.1 local0
  log 127.0.0.1 local1 notice

defaults
  log global
  mode http
  option httplog
  option dontlognull
  timeout connect 5000ms
  timeout client 50000ms
  timeout server 50000ms

listen stats
    bind 0.0.0.0:70
    stats enable
    stats uri /

frontend balancer
    bind 0.0.0.0:80
    mode http
    default_backend web_backends

backend web_backends
    mode http
    option forwardfor
    balance roundrobin
    server weba weba:80 check
    server webb webb:80 check
    server webc webc:80 check
    option httpchk GET /
    http-check expect status 200
```

docker-compose.yml

编写 docker-compose.yml 文件，这个是 Compose 使用的主模板文件。内容十分简单，指定 3 个 web 容器，以及 1 个 haproxy 容器。

```
weba:
    build: ./web
    expose:
        - 80

webb:
    build: ./web
    expose:
        - 80

webc:
    build: ./web
    expose:
        - 80

haproxy:
    image: haproxy:latest
    volumes:
        - ./haproxy:/haproxy-override
        - ./haproxy/haproxy.cfg:/usr/local/etc/haproxy/haproxy.c
fg:ro
    links:
        - weba
        - webb
        - webc
    ports:
        - "80:80"
        - "70:70"
    expose:
        - "80"
        - "70"
```

运行 compose 项目

现在 compose-haproxy-web 目录长成下面的样子。

```
compose-haproxy-web
├── docker-compose.yml
├── haproxy
│   └── haproxy.cfg
└── web
    ├── Dockerfile
    ├── index.html
    └── index.py
```

在该目录下执行 `docker-compose up` 命令，会整合输出所有容器的输出。

```
$sudo docker-compose up
Recreating composehaproxyweb_webb_1...
Recreating composehaproxyweb_webc_1...
Recreating composehaproxyweb_weba_1...
Recreating composehaproxyweb_haproxy_1...
Attaching to composehaproxyweb_webb_1, composehaproxyweb_webc_1,
 composehaproxyweb_weba_1, composehaproxyweb_haproxy_1
```

此时访问本地的 80 端口，会经过 haproxy 自动转发到后端的某个 web 容器上，刷新页面，可以观察到访问的容器地址的变化。

访问本地 70 端口，可以查看到 haproxy 的统计信息。

当然，还可以使用 consul、etcd 等实现服务发现，这样就可以避免手动指定后端的 web 容器了，更为灵活。

Compose 命令说明

命令对象与格式

对于 Compose 来说，大部分命令的对象既可以是项目本身，也可以指定为项目中的服务或者容器。如果没有特别的说明，命令对象将是项目，这意味着项目中所有的服务都会受到命令影响。

执行 `docker-compose [COMMAND] --help` 或者 `docker-compose help [COMMAND]` 可以查看具体某个命令的使用格式。

Compose 命令的基本的使用格式是

```
docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
```

命令选项

- `-f, --file FILE` 指定使用的 Compose 模板文件，默认为 `docker-compose.yml`，可以多次指定。
- `-p, --project-name NAME` 指定项目名称，默认将使用所在目录名称作为项目名。
- `--x-networking` 使用 Docker 的可拔插网络后端特性（需要 Docker 1.9 及以后版本）。
- `--x-network-driver DRIVER` 指定网络后端的驱动，默认为 `bridge`（需要 Docker 1.9 及以后版本）。
- `--verbose` 输出更多调试信息。
- `-v, --version` 打印版本并退出。

命令使用说明

`build`

格式为 `docker-compose build [options] [SERVICE...]`。

构建（重新构建）项目中的服务容器。

服务容器一旦构建后，将会带上一个标记名，例如对于 web 项目中的一个 db 容器，可能是 web_db。

可以随时在项目目录下运行 `docker-compose build` 来重新构建服务。

选项包括：

- `--force-rm` 删除构建过程中的临时容器。
- `--no-cache` 构建镜像过程中不使用 cache（这将加长构建过程）。
- `--pull` 始终尝试通过 pull 来获取更新版本的镜像。

`help`

获得一个命令的帮助。

`kill`

格式为 `docker-compose kill [options] [SERVICE...]`。

通过发送 `SIGKILL` 信号来强制停止服务容器。

支持通过 `-s` 参数来指定发送的信号，例如通过如下指令发送 `SIGINT` 信号。

```
$ docker-compose kill -s SIGINT
```

`logs`

格式为 `docker-compose logs [options] [SERVICE...]`。

查看服务容器的输出。默认情况下，docker-compose 将对不同的服务输出使用不同的颜色来区分。可以通过 `--no-color` 来关闭颜色。

该命令在调试问题的时候十分有用。

`pause`

格式为 `docker-compose pause [SERVICE...]`。

暂停一个服务容器。

`port`

格式为 `docker-compose port [options] SERVICE PRIVATE_PORT` 。

打印某个容器端口所映射的公共端口。

选项：

- `--protocol=proto` 指定端口协议，tcp（默认值）或者 udp。
- `--index=index` 如果同一服务存在多个容器，指定命令对象容器的序号（默认为 1）。

`ps`

格式为 `docker-compose ps [options] [SERVICE...]` 。

列出项目中目前的所有容器。

选项：

- `-q` 只打印容器的 ID 信息。

`pull`

格式为 `docker-compose pull [options] [SERVICE...]` 。

拉取服务依赖的镜像。

选项：

- `--ignore-pull-failures` 忽略拉取镜像过程中的错误。

`restart`

格式为 `docker-compose restart [options] [SERVICE...]` 。

重启项目中的服务。

选项：

- `-t, --timeout TIMEOUT` 指定重启前停止容器的超时（默认为 10 秒）。

`rm`

格式为 `docker-compose rm [options] [SERVICE...]` 。

删除所有（停止状态的）服务容器。推荐先执行 `docker-compose stop` 命令来停止容器。

选项：

- `-f, --force` 强制直接删除，包括非停止状态的容器。一般尽量不要使用该选项。
- `-v` 删除容器所挂载的数据卷。

`run`

格式为 `docker-compose run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]` 。

在指定服务上执行一个命令。

例如：

```
$ docker-compose run ubuntu ping docker.com
```

将会启动一个 ubuntu 服务容器，并执行 `ping docker.com` 命令。

默认情况下，如果存在关联，则所有关联的服务将会自动被启动，除非这些服务已经在运行中。

该命令类似启动容器后运行指定的命令，相关卷、链接等等都将会按照配置自动创建。

两个不同点：

- 给定命令将会覆盖原有的自动运行命令；
- 不会自动创建端口，以避免冲突。

如果不希望自动启动关联的容器，可以使用 `--no-deps` 选项，例如

```
$ docker-compose run --no-deps web python manage.py shell
```

将不会启动 web 容器所关联的其它容器。

选项：

- `-d` 后台运行容器。
- `--name NAME` 为容器指定一个名字。
- `--entrypoint CMD` 覆盖默认的容器启动指令。
- `-e KEY=VAL` 设置环境变量值，可多次使用选项来设置多个环境变量。
- `-u, --user=""` 指定运行容器的用户名或者 uid。
- `--no-deps` 不自动启动关联的服务容器。
- `--rm` 运行命令后自动删除容器，`d` 模式下将忽略。
- `-p, --publish=[]` 映射容器端口到本地主机。
- `--service-ports` 配置服务端口并映射到本地主机。
- `-T` 不分配伪 tty，意味着依赖 tty 的指令将无法运行。

`scale`

格式为 `docker-compose scale [options] [SERVICE=NUM...]` 。

设置指定服务运行的容器个数。

通过 `service=num` 的参数来设置数量。例如：

```
$ docker-compose scale web=3 db=2
```

将启动 3 个容器运行 web 服务，2 个容器运行 db 服务。

一般的，当指定数目多于该服务当前实际运行容器，将新创建并启动容器；反之，将停止容器。

选项：

- `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。

`start`

格式为 `docker-compose start [SERVICE...]` 。

启动已经存在的服务容器。

`stop`

格式为 `docker-compose stop [options] [SERVICE...]` 。

停止已经处于运行状态的容器，但不删除它。通过 `docker-compose start` 可以再次启动这些容器。

选项：

- `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。

unpause

格式为 `docker-compose unpause [SERVICE...]` 。

恢复处于暂停状态中的服务。

up

格式为 `docker-compose up [options] [SERVICE...]` 。

该命令十分强大，它将尝试自动完成包括构建镜像，（重新）创建服务，启动服务，并关联服务相关容器的一系列操作。

链接的服务都将会被自动启动，除非已经处于运行状态。

可以说，大部分时候都可以直接通过该命令来启动一个项目。

默认情况， `docker-compose up` 启动的容器都在前台，控制台将会同时打印所有容器的输出信息，可以很方便进行调试。

当通过 `Ctrl-C` 停止命令时，所有容器将会停止。

如果使用 `docker-compose up -d` ，将会在后台启动并运行所有的容器。一般推荐生产环境下使用该选项。

默认情况，如果服务容器已经存在， `docker-compose up` 将会尝试停止容器，然后重新创建（保持使用 `volumes-from` 挂载的卷），以保证新启动的服务匹配 `docker-compose.yml` 文件的最新内容。如果用户不希望容器被停止并重新创建，可以使用 `docker-compose up --no-recreate` 。这样将只会启动处于停止状态的容器，而忽略已经运行的服务。如果用户只想重新部署某个服务，可以使用 `docker-compose up --no-deps -d <SERVICE_NAME>` 来重新创建服务并后台停止旧服务，启动新服务，并不会影响到其所依赖的服务。

选项：

- `-d` 在后台运行服务容器。

* `--no-color` 不使用颜色来区分不同的服务的控制台输出。
* `--no-deps` 不启动服务所链接的容器。
* `--force-recreate` 强制重新创建容器，不能与 `--no-recreate` 同时使用。
* `--no-recreate` 如果容器已经存在了，则不重新创建，不能与 `--force-recreate` 同时使用。
* `--no-build` 不自动构建缺失的服务镜像。
* `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。

`migrate-to-labels`

格式为 `docker-compose migrate-to-labels` 。

重新创建容器，并添加 label。

主要用于升级 1.2 及更早版本中创建的容器，添加缺失的容器标签。

实际上，最彻底的办法当然是删除项目，然后重新创建。

`version`

格式为 `docker-compose version` 。

打印版本信息。

Compose 模板文件

模板文件是使用 Compose 的核心，涉及到的指令关键字也比较多。但大家不用担心，这里面大部分指令跟 `docker run` 相关参数的含义都是类似的。

默认的模板文件名称为 `docker-compose.yml` ，格式为 YAML 格式。

在旧版本（版本 1）中，其中每个顶级元素为服务名称，次级元素为服务容器的配置信息，例如

```
webapp:
  image: examples/web
  ports:
    - "80:80"
  volumes:
    - "/data"
```

版本 2 扩展了 Compose 的语法，同时尽量保持跟版本 1 的兼容，除了可以声明网络和存储信息外，最大的不同一是添加了版本信息，另一个是需要将所有的服务放到 `services` 根下面。

例如，上面例子改写为版本 2，内容为

```
version: "2"
services:
  webapp:
    image: examples/web
    ports:
      - "80:80"
    volumes:
      - "/data"
```

注意每个服务都必须通过 `image` 指令指定镜像或 `build` 指令（需要 Dockerfile）等来自动构建生成镜像。

如果使用 `build` 指令，在 `Dockerfile` 中设置的选项(例如：`CMD`，`EXPOSE`，`VOLUME`，`ENV` 等) 将会自动被获取，无需在 `docker-compose.yml` 中再次设置。

下面分别介绍各个指令的用法。

`build`

指定 `Dockerfile` 所在文件夹的路径（可以是绝对路径，或者相对 docker-compose.yml 文件的路径）。 `Compose` 将会利用它自动构建这个镜像，然后使用这个镜像。

```
build: /path/to/build/dir
```

`cap_add, cap_drop`

指定容器的内核能力（capacity）分配。

例如，让容器拥有所有能力可以指定为：

```
cap_add:
  - ALL
```

去掉 NET_ADMIN 能力可以指定为：

```
cap_drop:
  - NET_ADMIN
```

`command`

覆盖容器启动后默认执行的命令。

```
command: echo "hello world"
```

`cgroup_parent`

指定父 cgroup 组，意味着将继承该组的资源限制。

例如，创建了一个 cgroup 组名称为 `cgroups_1` 。

```
cgroup_parent: cgroups_1
```

`container_name`

指定容器名称。默认将会使用 `项目名称_服务名称_序号` 这样的格式。

例如：

```
container_name: docker-web-container
```

需要注意，指定容器名称后，该服务将无法进行扩展（scale），因为 Docker 不允许多个容器具有相同的名称。

`devices`

指定设备映射关系。

例如：

```
devices:
  - "/dev/ttyUSB1:/dev/ttyUSB0"
```

`dns`

自定义 DNS 服务器。可以是一个值，也可以是一个列表。

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

`dns_search`

配置 DNS 搜索域。可以是一个值，也可以是一个列表。

```
dns_search: example.com
dns_search:
  - domain1.example.com
  - domain2.example.com
```

`dockerfile`

如果需要指定额外的编译镜像的 Dockefile 文件，可以通过该指令来指定。

例如

```
dockerfile: Dockerfile-alternate
```

注意，该指令不能跟 `image` 同时使用，否则 Compose 将不知道根据哪个指令来生成最终的服务镜像。

`env_file`

从文件中获取环境变量，可以为单独的文件路径或列表。

如果通过 `docker-compose -f FILE` 方式来指定 Compose 模板文件，则 `env_file` 中变量的路径会基于模板文件路径。

如果有变量名称与 `environment` 指令冲突，则按照惯例，以后者为准。

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

环境变量文件中每一行必须符合格式，支持 `#` 开头的注释行。

```
# common.env: Set development environment
PROG_ENV=development
```

`environment`

设置环境变量。你可以使用数组或字典两种格式。

只给定名称的变量会自动获取运行 Compose 主机上对应变量的值，可以用来防止泄露不必要的数据。

例如

```
environment:
  RACK_ENV: development
  SESSION_SECRET:
```

或者

```
environment:
  - RACK_ENV=development
  - SESSION_SECRET
```

注意，如果变量名称或者值中用到 `true|false，yes|no` 等表达布尔含义的词汇，最好放到引号里，避免 YAML 自动解析某些内容为对应的布尔语义。

`http://yaml.org/type/bool.html` 中给出了这些特定词汇，包括

```
 y|Y|yes|Yes|YES|n|N|no|No|NO
|true|True|TRUE|false|False|FALSE
|on|On|ON|off|Off|OFF
```

`expose`

暴露端口，但不映射到宿主机，只被连接的服务访问。

仅可以指定内部端口为参数

```
expose:
 - "3000"
 - "8000"
```

`extends`

基于其它模板文件进行扩展。

例如我们已经有了一个 webapp 服务，定义一个基础模板文件为 `common.yml`。

```
# common.yml
webapp:
  build: ./webapp
  environment:
    - DEBUG=false
    - SEND_EMAILS=false
```

再编写一个新的 `development.yml` 文件，使用 `common.yml` 中的 webapp 服务进行扩展。

```
# development.yml
web:
  extends:
    file: common.yml
    service: webapp
  ports:
    - "8000:8000"
  links:
    - db
  environment:
    - DEBUG=true
db:
  image: postgres
```

后者会自动继承 common.yml 中的 webapp 服务及环境变量定义。

使用 extends 需要注意：

- 要避免出现循环依赖，例如 `A 依赖 B，B 依赖 C，C 反过来依赖 A` 的情况。
- extends 不会继承 links 和 volumes_from 中定义的容器和数据卷资源。

一般的，推荐在基础模板中只定义一些可以共享的镜像和环境变量，在扩展模板中具体指定应用变量、链接、数据卷等信息。

`external_links`

链接到 docker-compose.yml 外部的容器，甚至 并非 `Compose` 管理的外部容器。参数格式跟 `links` 类似。

```
external_links:
  - redis_1
  - project_db_1:mysql
  - project_db_1:postgresql
```

`extra_hosts`

类似 Docker 中的 `--add-host` 参数，指定额外的 host 名称映射信息。

例如：

```
extra_hosts:
  - "googledns:8.8.8.8"
  - "dockerhub:52.1.157.61"
```

会在启动后的服务容器中 `/etc/hosts` 文件中添加如下两条条目。

```
8.8.8.8 googledns
52.1.157.61 dockerhub
```

`image`

指定为镜像名称或镜像 ID。如果镜像在本地不存在，`Compose` 将会尝试拉去这个镜像。

例如：

```
image: ubuntu
image: orchardup/postgresql
image: a4bc65fd
```

`labels`

为容器添加 Docker 元数据（metadata）信息。例如可以为容器添加辅助说明信息。

```
labels:
  com.startupteam.description: "webapp for a startup team"
  com.startupteam.department: "devops department"
  com.startupteam.release: "rc3 for v1.0"
```

`links`

链接到其它服务中的容器。使用服务名称（同时作为别名）或服务名称：服务别名 `（SERVICE:ALIAS）` 格式都可以。

```
links:
 - db
 - db:database
 - redis
```

使用的别名将会自动在服务容器中的 `/etc/hosts` 里创建。例如：

```
172.17.2.186  db
172.17.2.186  database
172.17.2.187  redis
```

被链接容器中相应的环境变量也将被创建。

`log_driver`

类似 Docker 中的 `--log-driver` 参数，指定日志驱动类型。

目前支持三种日志驱动类型。

```
log_driver: "json-file"
log_driver: "syslog"
log_driver: "none"
```

`log_opt`

日志驱动的相关参数。

例如

```
log_driver: "syslog"
log_opt:
  syslog-address: "tcp://192.168.0.42:123"
```

`net`

设置网络模式。使用和 `docker client` 的 `--net` 参数一样的值。

```
net: "bridge"
net: "none"
net: "container:[name or id]"
net: "host"
```

`pid`

跟主机系统共享进程命名空间。打开该选项的容器之间，以及容器和宿主机系统之间可以通过进程 ID 来相互访问和操作。

```
pid: "host"
```

`ports`

暴露端口信息。

使用宿主：容器 `（HOST:CONTAINER）` 格式，或者仅仅指定容器的端口（宿主将会随机选择端口）都可以。

```
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
```

注意：当使用 *`HOST:CONTAINER`* 格式来映射端口时，如果你使用的容器端口小于 *60* 并且没放到引号里，可能会得到错误结果，因为 *`YAML`* 会自动解析 *`xx:yy`* 这种数字格式为 *60* 进制。为避免出现这种问题，建议数字串都采用引号包括起来的字符串格式。

`security_opt`

指定容器模板标签（label）机制的默认属性（用户、角色、类型、级别等）。

例如配置标签的用户名和角色名。

```
security_opt:
    - label:user:USER
    - label:role:ROLE
```

`ulimits`

指定容器的 ulimits 限制值。

例如，指定最大进程数为 65535，指定文件句柄数为 20000（软限制，应用可以随时修改，不能超过硬限制） 和 40000（系统硬限制，只能 root 用户提高）。

```
  ulimits:
    nproc: 65535
    nofile:
      soft: 20000
      hard: 40000
```

`volumes`

数据卷所挂载路径设置。可以设置宿主机路径 （`HOST:CONTAINER`） 或加上访问模式 （`HOST:CONTAINER:ro`）。

该指令中路径支持相对路径。例如

```
volumes:
 - /var/lib/mysql
 - cache/:/tmp/cache
 - ~/configs:/etc/configs/:ro
```

`volumes_driver`

较新版本的 Docker 支持数据卷的插件驱动。

用户可以先使用第三方驱动创建一个数据卷，然后使用名称来访问它。

此时，可以通过 `volumes_driver` 来指定驱动。

```
volume_driver: mydriver
```

`volumes_from`

从另一个服务或容器挂载它的数据卷。

```
volumes_from:
 - service_name
 - container_name
```

其它指令

此外，还有包括 `cpu_shares, cpuset, domainname, entrypoint, hostname, ipc, mac_address, mem_limit, memswap_limit, privileged, read_only, restart, stdin_open, tty, user, working_dir` 等指令，基本跟 docker-run 中对应参数的功能一致。

例如，指定使用 cpu 核 0 和 核 1，只用 50% 的 CPU 资源：

```
cpu_shares: 73
cpuset: 0,1
```

指定服务容器启动后执行的命令。

```
entrypoint: /code/entrypoint.sh
```

指定容器中运行应用的用户名。

```
user: nginx
```

指定容器中工作目录。

```
working_dir: /code
```

指定容器中搜索域名、主机名、mac 地址等。

```
domainname: your_website.com
hostname: test
mac_address: 08-00-27-00-0C-0A
```

指定容器中

```
ipc: host
```

指定容器中内存和内存交换区限制都为 1G。

```
mem_limit: 1g
memswap_limit: 1g
```

允许容器中运行一些特权命令。

```
privileged: true
```

指定容器退出后的重启策略为始终重启。该命令对保持服务始终运行十分有效，在生产环境中推荐配置为 `always` 或者 `unless-stopped` 。

```
restart: always
```

以只读模式挂载容器的 root 文件系统，意味着不能对容器内容进行修改。

```
read_only: true
```

打开标准输入，可以接受外部输入。

```
stdin_open: true
```

模拟一个假的远程控制台。

```
tty: true
```

读取环境变量

从 1.5.0 版本开始，Compose 模板文件支持动态读取主机的系统环境变量。

例如，下面的 Compose 文件将从运行它的环境中读取变量 ${MONGO_VERSION} 的值，并写入执行的指令中。

```
db:
  image: "mongo:${MONGO_VERSION}"
```

如果执行 `MONGO_VERSION=3.0 docker-compose up` 则会启动一个 `mongo:3.2` 镜像的容器；如果执行 `MONGO_VERSION=2.8 docker-compose up` 则会启动一个 `mongo:2.8` 镜像的容器。

使用 Django

我们现在将使用 Compose 配置并运行一个 Django/PostgreSQL 应用。在此之前，先确保 Compose 已经 安装。

在一切工作开始前，需要先设置好三个必要的文件。
第一步，因为应用将要运行在一个满足所有环境依赖的 Docker 容器里面，那么我们可以通过编辑 `Dockerfile` 文件来指定 Docker 容器要安装内容。内容如下：

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
ADD requirements.txt /code/
RUN pip install -r requirements.txt
ADD . /code/
```

以上内容指定应用将使用安装了 Python 以及必要依赖包的镜像。更多关于如何编写 Dockerfile 文件的信息可以查看 镜像创建 和 Dockerfile 使用。

第二步，在 `requirements.txt` 文件里面写明需要安装的具体依赖包名 。

```
Django
psycopg2
```

就是这么简单。
第三步，`docker-compose.yml` 文件将把所有的东西关联起来。它描述了应用的构成（一个 web 服务和一个数据库）、使用的 Docker 镜像、镜像之间的连接、挂载到容器的卷，以及服务开放的端口。

```
db:
  image: postgres
web:
  build: .
  command: python manage.py runserver 0.0.0.0:8000
  volumes:
    - .:/code
  ports:
    - "8000:8000"
  links:
    - db
```

查看 `docker-compose.yml` 章节 了解更多详细的工作机制。

现在我们就可以使用 `docker-compose run` 命令启动一个 Django 应用了。

```
$ docker-compose run web django-admin.py startproject django_exa
mple .
```

Compose 会先使用 `Dockerfile` 为 web 服务创建一个镜像，接着使用这个镜像在容器里运行 `django-admin.py startproject django_example .` 指令。

这将在当前目录生成一个 Django 应用。

```
$ ls
Dockerfile       docker-compose.yml          django_example
  manage.py        requirements.txt
```

首先，我们要为应用设置好数据库的连接信息。用以下内容替换 `django_example/settings.py` 文件中 `DATABASES = ...` 定义的节点内容。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

这些信息是在 postgres Docker 镜像固定设置好的。
然后，运行 `docker-compose up` ：

```
Recreating myapp_db_1...
Recreating myapp_web_1...
Attaching to myapp_db_1, myapp_web_1
myapp_db_1 |
myapp_db_1 | PostgreSQL stand-alone backend 9.1.11
myapp_db_1 | 2014-01-27 12:17:03 UTC LOG:  database system is re
ady to accept connections
myapp_db_1 | 2014-01-27 12:17:03 UTC LOG:  autovacuum launcher s
tarted
myapp_web_1 | Validating models...
myapp_web_1 |
myapp_web_1 | 0 errors found
myapp_web_1 | January 27, 2014 - 12:12:40
myapp_web_1 | Django version 1.6.1, using settings 'django_examp
le.settings'
myapp_web_1 | Starting development server at http://0.0.0.0:8000
/
myapp_web_1 | Quit the server with CONTROL-C.
```

这个 web 应用已经开始在你的 docker 守护进程里监听着 5000 端口了（如果你有使用 boot2docker ，执行 `boot2docker ip` ，就会看到它的地址）。

你还可以在 Docker 上运行其它的管理命令，例如对于同步数据库结构这种事，在运行完 `docker-compose up` 后，在另外一个终端运行以下命令即可：

```
$ docker-compose run web python manage.py syncdb
```

使用 Rail

我们现在将使用 Compose 配置并运行一个 Rails/PostgreSQL 应用。在开始之前，先确保 Compose 已经 安装。

在一切工作开始前，需要先设置好三个必要的文件。
首先，因为应用将要运行在一个满足所有环境依赖的 Docker 容器里面，那么我们可以通过编辑 `Dockerfile` 文件来指定 Docker 容器要安装内容。内容如下：

```
FROM ruby
RUN apt-get update -qq && apt-get install -y build-essential lib
pq-dev
RUN mkdir /myapp
WORKDIR /myapp
ADD Gemfile /myapp/Gemfile
RUN bundle install
ADD . /myapp
```

以上内容指定应用将使用安装了 Ruby、Bundler 以及其依赖件的镜像。更多关于如何编写 Dockerfile 文件的信息可以查看 镜像创建 和 Dockerfile 使用。 下一步，我们需要一个引导加载 Rails 的文件 `Gemfile` 。 等一会儿它还会被 `rails new` 命令覆盖重写。

```
source 'https://rubygems.org'
gem 'rails', '4.0.2'
```

最后，`docker-compose.yml` 文件才是最神奇的地方。 `docker-compose.yml` 文件将把所有的东西关联起来。它描述了应用的构成（一个 web 服务和一个数据库）、每个镜像的来源（数据库运行在使用预定义的 PostgreSQL 镜像，web 应用侧将从本地目录创建）、镜像之间的连接，以及服务开放的端口。

```
db:
  image: postgres
  ports:
    - "5432"
web:
  build: .
  command: bundle exec rackup -p 3000
  volumes:
    - .:/myapp
  ports:
    - "3000:3000"
  links:
    - db
```

所有文件就绪后，我们就可以通过使用 `docker-compose run` 命令生成应用的骨架了。

```
$ docker-compose run web rails new . --force --database=postgres
ql --skip-bundle
```

Compose 会先使用 `Dockerfile` 为 web 服务创建一个镜像，接着使用这个镜像在容器里运行 `rails new` 和它之后的命令。一旦这个命令运行完后，应该就可以看一个崭新的应用已经生成了。

```
$ ls
Dockerfile   app            docker-compose.yml      tmp
Gemfile      bin            lib            vendor
Gemfile.lock condocker-compose        log
README.rdoc  condocker-compose.ru     public
Rakefile     db             test
```

在新的 `Gemfile` 文件去掉加载 `therubyracer` 的行的注释，这样我们便可以使用 Javascript 运行环境：

```
gem 'therubyracer', platforms: :ruby
```

现在我们已经有一个新的 `Gemfile` 文件，需要再重新创建镜像。（这个会步骤会改变 Dockerfile 文件本身，仅仅需要重建一次）。

```
$ docker-compose build
```

应用现在就可以启动了，但配置还未完成。Rails 默认读取的数据库目标是 `localhost` ，我们需要手动指定容器的 `db` 。同样的，还需要把用户名修改成和 postgres 镜像预定的一致。 打开最新生成的 `database.yml` 文件。用以下内容替换：

```
development: &default
  adapter: postgresql
  encoding: unicode
  database: postgres
  pool: 5
  username: postgres
  password:
  host: db

test:
  <<: *default
  database: myapp_test
```

现在就可以启动应用了。

```
$ docker-compose up
```

如果一切正常，你应该可以看到 PostgreSQL 的输出，几秒后可以看到这样的重复信息：

```
myapp_web_1 | [2014-01-17 17:16:29] INFO  WEBrick 1.3.1
myapp_web_1 | [2014-01-17 17:16:29] INFO  ruby 2.0.0 (2013-11-22
) [x86_64-linux-gnu]
myapp_web_1 | [2014-01-17 17:16:29] INFO  WEBrick::HTTPServer#st
art: pid=1 port=3000
```

最后， 我们需要做的是创建数据库，打开另一个终端，运行：

```
$ docker-compose run web rake db:create
```

这个 web 应用已经开始在你的 docker 守护进程里面监听着 3000 端口了（如果你有使用 boot2docker ，执行 `boot2docker ip` ，就会看到它的地址）。



使用 Wordpress

Compose 让 Wordpress 运行在一个独立的环境中很简易。

安装 Compose ，然后下载 Wordpress 到当前目录：

```
wordpress.org/latest.tar.gz | tar -xvzf -
```

这将会创建一个叫 wordpress 目录，你也可以重命名成你想要的名字。在目录里面，创建一个 `Dockerfile` 文件，定义应用的运行环境：

```
FROM orchardup/php5
ADD . /code
```

以上内容告诉 Docker 创建一个包含 PHP 和 Wordpress 的镜像。更多关于如何编写 Dockerfile 文件的信息可以查看 镜像创建 和 Dockerfile 使用。

下一步， `docker-compose.yml` 文件将开启一个 web 服务和一个独立的 MySQL 实例：

```
web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  ports:
    - "8000:8000"
  links:
    - db
  volumes:
    - .:/code
db:
  image: orchardup/mysql
  environment:
    MYSQL_DATABASE: wordpress
```

要让这个应用跑起来还需要两个文件。 第一个， `wp-condocker-compose.php` ，它是一个标准的 Wordpress 配置文件，有一点需要修改的是把数据库的配置指向 `db` 容器。

```
<?php
define('DB_NAME', 'wordpress');
define('DB_USER', 'root');
define('DB_PASSWORD', '');
define('DB_HOST', "db:3306");
define('DB_CHARSET', 'utf8');
define('DB_COLLATE', '');

define('AUTH_KEY',         'put your unique phrase here');
define('SECURE_AUTH_KEY',  'put your unique phrase here');
define('LOGGED_IN_KEY',    'put your unique phrase here');
define('NONCE_KEY',        'put your unique phrase here');
define('AUTH_SALT',        'put your unique phrase here');
define('SECURE_AUTH_SALT', 'put your unique phrase here');
define('LOGGED_IN_SALT',   'put your unique phrase here');
define('NONCE_SALT',       'put your unique phrase here');

$table_prefix  = 'wp_';
define('WPLANG', '');
define('WP_DEBUG', false);

if ( !defined('ABSPATH') )
    define('ABSPATH', dirname(__FILE__) . '/');

require_once(ABSPATH . 'wp-settings.php');
```

第二个， `router.php` ，它告诉 PHP 内置的服务器怎么运行 Wordpress:

```
<?php

$root = $_SERVER['DOCUMENT_ROOT'];
chdir($root);
$path = '/'.ltrim(parse_url($_SERVER['REQUEST_URI'])['path'],'/'
);
set_include_path(get_include_path().':'.__DIR__);
if(file_exists($root.$path))
{
    if(is_dir($root.$path) && substr($path,strlen($path) - 1, 1)
 !== '/')
        $path = rtrim($path,'/').'/index.php';
    if(strpos($path,'.php') === false) return false;
    else {
        chdir(dirname($root.$path));
        require_once $root.$path;
    }
}else include_once 'index.php';
```

这些配置文件就绪后，在你的 Wordpress 目录里面执行 `docker-compose up` 指令，Compose 就会拉取镜像再创建我们所需要的镜像，然后启动 web 和数据库容器。 接着访问 docker 守护进程监听的 8000 端口就能看你的 Wordpress 网站了。（如果你有使用 boot2docker ，执行 `boot2docker ip` ，就会看到它的地址）。

Docker Machine 项目

Docker Machine 是 Docker 官方编排（Orchestration）项目之一，负责在多种平台上快速安装 Docker 环境。

本章将介绍 Machine 项目情况以及安装和使用。

简介



图 *1.16.1.1 - Docker Machine*

Docker Machine 项目基于 Go 语言实现，目前在 Github 上进行维护。

技术讨论 IRC 频道为 `#docker-machine` 。

安装

Docker Machine 可以在多种操作系统平台上安装，包括 Linux、Mac OS，以及 Windows。

Linux/Mac OS

在 Linux/Mac OS 上的安装十分简单，推荐从 官方 Release 库 直接下载编译好的二进制文件即可。

例如，在 Linux 64 位系统上直接下载对应的二进制包。

```
$ sudo curl -L https://github.com/docker/machine/releases/download/v0.3.1-rc1/docker-machine_linux-amd64 > /usr/local/bin/docker-machine
$ chmod +x /usr/local/bin/docker-machine
```

完成后，查看版本信息，验证运行正常。

```
$ docker-machine -v
docker-machine version 0.3.1-rc1 (993f2db)
```

Windows

Windows 下面要复杂一些，首先需要安装 msysgit。

msysgit 是 Windows 下的 git 客户端软件包，会提供类似 Linux 下的一些基本的工具，例如 ssh 等。

安装之后，启动 msysgit 的命令行界面，仍然通过下载二进制包进行安装，需要下载 docker 客户端和 docker-machine。

```
$ curl -L https://get.docker.com/builds/Windows/x86_64/docker-la
test.exe > /bin/docker

$ curl -L https://github.com/docker/machine/releases/download/v0
.3.1-rc1/docker-machine_windows-amd64.exe > /bin/docker-machine
```

使用

Docker Machine 支持多种后端驱动，包括虚拟机、本地主机和云平台等。

本地主机实例

首先确保本地主机可以通过 user 账号的 key 直接 ssh 到目标主机。

使用 generic 类型的驱动，创建一台 Docker 主机，命名为 test。

```
$ docker-machine create -d generic --generic-ip-address=10.0.100
.101 --generic-ssh-user=user test
```

创建主机成功后，可以通过 env 命令来让后续操作对象都是目标主机。

```
$ docker-machine env test
```

支持驱动

通过 `-d` 选项可以选择支持的驱动类型。

- amazonec2
- azure
- digitalocean
- exoscale
- generic
- google
- none
- openstack
- rackspace
- softlayer
- virtualbox
- vmwarevcloudair
- vmwarevsphere

操作命令

- `active` 查看活跃的 Docker 主机
- `config` 输出连接的配置信息
- `create` 创建一个 Docker 主机
- `env` 显示连接到某个主机需要的环境变量
- `inspect` 输出主机更多信息
- `ip` 获取主机地址
- `kill` 停止某个主机
- `ls` 列出所有管理的主机
- `regenerate-certs` 为某个主机重新生成 TLS 认证信息
- `restart` 重启主机
- `rm` 删除某台主机
- `ssh` SSH 到主机上执行命令
- `scp` 在主机之间复制文件
- `start` 启动一个主机
- `stop` 停止一个主机
- `upgrade` 更新主机 Docker 版本为最新
- `url` 获取主机的 URL
- `help, h` 输出帮助信息

每个命令，又带有不同的参数，可以通过

```
docker-machine <COMMAND> -h
```

来查看具体的用法。

Docker 三剑客之 Docker Swarm

Docker Swarm 是 Docker 官方三剑客项目之一，提供 Docker 容器集群服务，是 Docker 官方对容器云生态进行支持的核心方案。

使用它，用户可以将多个 Docker 主机封装为单个大型的虚拟 Docker 主机，快速打造一套容器云平台。

本章将介绍 Swarm 项目的相关情况，以及如何进行安装和使用。最后还对 Swarm 的服务发现后端、调度器和过滤器等功能进行讲解。

Swarm 简介



图 *1.17.1.1 - Docker Swarm*

Docker Swarm 是 Docker 公司推出的官方容器集群平台，基于 Go 语言实现，代码开源在 https://github.com/docker/swarm。目前，包括 Rackspace 等平台都采用了 Swarm，用户也很容易在 AWS 等公有云平台使用 Swarm。

Swarm 的前身是 Beam 项目和 libswarm 项目，首个正式版本（Swarm V1）在 2014 年 12 月初发布。为了提高可扩展性，2016 年 2 月对架构进行重新设计，推出了 V2 版本，支持超过 1K 个节点。最新的 Docker Engine 已经集成了 SwarmKit，加强了对 Swarm 的协作支持。

作为容器集群管理器，Swarm 最大的优势之一就是 100% 支持标准的 Docker API。各种基于标准 API 的工具比如 Compose、docker-py、各种管理软件，甚至 Docker 本身等都可以很容易的与 Swarm 进行集成。这大大方便了用户将原先基于单节点的系统移植到 Swarm 上。同时 Swarm 内置了对 Docker 网络插件的支持，用户可以很容易地部署跨主机的容器集群服务。

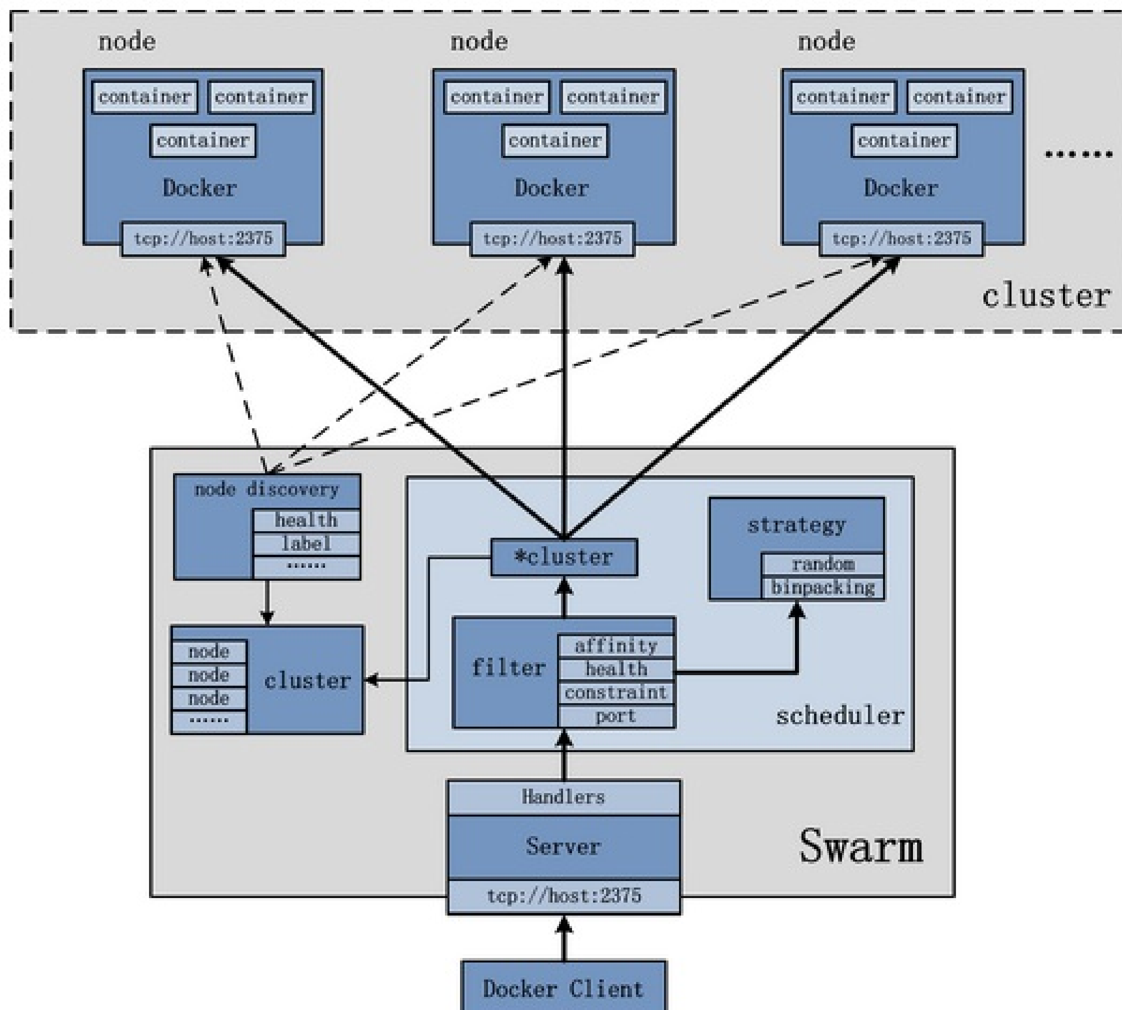


图 *1.17.1.2 - Swarm* 基本结构图

上图是来自官方的 V1 结构图。可以看出，Swarm 是典型的 master-slave 结构，通过发现服务来选举 manager。manager 是中心管理节点，各个 node 上运行 agent 接受 manager 的统一管理。

在 V2 中，集群中会自动通过 Raft 协议分布式选举出 manager 节点，无需额外的发现服务支持，避免了单点瓶颈。同时，V2 中内置了基于 DNS 的负载均衡和对外部负载均衡机制的集成支持。

目前，Swarm V1 支持 Docker 版本为 1.6.0+，V2 支持 Docker 版本为 1.12.0+。本章将以 Swarm V1 为主进行介绍，并结合 V2 的部分最新特性。

安装 Swarm

Swarm 安装有几种方式，可以基于 Docker Machine 来进行安装，也可以手动配置。为了能更容易理解 Swarm 的组件和更灵活的进行管理，推荐使用手动配置方式。

对于 Docker 1.12+ 版本，Swarm 相关命令已经原生嵌入到了 Docker engine 的支持，对于较低版本的 Docker，需要额外进行配置。

下载镜像

Docker 官方已经提供了 Swarm 镜像使用，需要在所有被 Swarm 管理的 Docker 主机上下载该镜像。

```
$ docker pull swarm
```

可以使用下面的命令来查看 Swarm 版本，验证是否成功下载 Swarm 镜像。

```
$ docker run --rm swarm -v
swarm version 1.2.2 (34e3da3)
```

配置节点

Docker 主机在加入 Swarm 集群前，需要进行一些简单配置，添加 Docker daemon 的网络监听。

例如，在启动 Docker daemon 的时候通过 `-H` 参数：

```
$ sudo docker daemon -H tcp://0.0.0.0:2375 -H unix:///var/run/do
cker.sock
```

注：*Docker 1.8.0* 版本之前不支持 *daemon* 命令，可以用 *-d* 代替。

如果是通过服务方式启动，则需要修改服务的配置文件。

以 Ubuntu 14.04 为例，配置文件为 `/etc/default/docker` （其他版本的 Linux 上略有不同）。

在文件的最后添加：

```
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2375 -H unix:///var/r
un/docker.sock"
```

启动集群

Docker 集群管理需要使用服务发现（Service Discover）功能，Swarm 支持以下的几种方式：DockerHub、本地文件、etcd、consul、zookeeper 和手动指定节点 IP 地址信息等。

除了手动指定外，这些方法原理上都是通过维护一套数据库机制，来管理集群中注册节点的 Docker daemon 的访问信息。

本地配置集群推荐使用 consul 作为服务发现后端。利用社区提供的 Docker 镜像，整个过程只需要三步即可完成。

启动 Consul 服务后端

启动 consul 服务容器，映射到主机的 8500 端口。

```
$ docker run -d -p 8500:8500 --name=consul progrium/consul -serv
er -bootstrap
```

获取到本地主机的地址作为 consul 的服务地址： `<consul_ip>:8500` 。

启动管理节点

首先，启动一个主管理节点，映射到主机的 4000 端口，并获取所在主机地址为 `<manager0_ip>` 。其中 4000 端口是 Swarm 管理器的默认监听端口，用户也可以指定映射为其它端口。

```
$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication
 --advertise <manager0_ip>:4000 consul://<consul_ip>:8500
```

为了提高高可用性，用户也可以启动从管理节点。假定获取所在主机地址为 `<manager1_ip>` 。

```
$ docker run -d swarm manage -H :4000 --replication --advertise
<manager1_ip>:4000 consul://<consul_ip>:8500
```

启动工作节点

需要在每个工作节点上启动 agent 服务。

获取节点的主机地址为 `<node_ip>` ，并指定前面获取到的 consul 服务地址。

```
$ docker run -d swarm join --advertise=<node_ip>:2375 consul://<
consul_ip>:8500
```

节点启动后，用户可以指定 Docker 服务地址为 `<manager0_ip>:4000>` 来测试各种 Docker 命令，可以看到整个 Swarm 集群就像一个虚拟的 Docker 主机一样正常工作。

由于 Swarm 实际上是通过 agent 调用了本地的 Docker daemon 来运行容器，当 Swarm 集群服务出现故障时，无法接受新的请求，但已经运行起来的容器将不会受到影响。

使用 Swarm

前面演示了基于 consul 服务发现后端来配置一个本地 Swarm 集群。其中，consul 也可以被替换为 etcd、zookeeper 等。

另外一个更方便的方式是直接使用 DockerHub 提供的免费服务发现后端。

下面使用这种方式来演示 Swarm 的主要操作，包括：

- create：创建一个集群；
- list：列出集群中的节点；
- manage：管理一个集群；
- join：让节点加入到某个集群。

注意，使用 DockerHub 的服务发现后端，需要各个节点能通过公网访问到 DockerHub 的服务接口。

创建集群 id

在任意一台安装了 Swarm 的机器上执行 `swarm create` 命令来在 DockerHub 服务上进行注册。

Swarm 会通过服务发现后端（此处为 DockerHub 提供）来获取一个唯一的由数字和字母组成的 token，用来标识要管理的集群。

```
$ docker run --rm swarm create
946d65606f7c2f49766e4dddac5b4365
```

注意返回的字符串，这是集群的唯一 id，加入集群的各个节点将需要这个信息。

配置集群节点

在所有要加入集群的普通节点上面执行 `swarm join` 命令，表示把这台机器加入指定集群当中。

例如某台机器 IP 地址为 `192.168.0.2`，将其加入我们刚创建的 `946d65606f7c2f49766e4dddac5b4365` 集群，则可以通过：

```
$ docker run --rm swarm join --addr=192.168.0.2:2375 token://946d65606f7c2f49766e4dddac5b4365
time="2015-12-09T08:59:43Z" level=info msg="Registering on the discovery service every 20s..." addr="192.168.0.2:2375" discovery="token://946d65606f7c2f49766e4dddac5b4365"
...
```

注：其中 `--addr` 指定的 *IP* 地址信息将被发送给服务发现后端，用以区分集群不同的节点。*manager* 服务必须要通过这个地址可以访问到该节点。

通过控制台可以看到，上述命令执行后，默认每隔 20 秒（可以通过 `--heartbeat` 选项指定），会输出一条心跳信息。对于发现服务后端来说，默认如果超过 60 秒（可以通过 `--ttl` 选项指定）没有收到心跳信息，则将节点从列表中删除。

如果不希望看到输出日志信息，则可以用 `-d` 选项替换 `--rm` 选项，让服务后台执行。

执行 `swarm join` 命令实际上是通过 agent 把自己的信息注册到发现服务上，因此，此时对于后端的发现服务来说，已经可以看到有若干节点注册上来了。那么，如何管理和使用这些节点呢，这就得需要 Swarm 的 manager 服务了。

配置管理节点

配置管理节点需要通过 `swarm manage` 命令，该命令将启动 manager 服务，默认监听到 `2375` 端口，所有对集群的管理可以通过该服务接口进行。

读者可能注意到，manager 服务默认监听的端口跟 Docker 服务监听端口是一样的，这是为了兼容其它基于 Docker 的服务，可以无缝地切换到 Swarm 平台上来。

仍然在节点 `192.168.0.2` 进行操作。由于我们是采用 Docker 容器形式启动 manager 服务，本地的 `2375` 端口已经被 Docker Daemon 占用。我们将 manager 服务监听端口映射到本地一个空闲的 `12375` 端口。

```
$ docker run -d -p 12375:2375 swarm manage token://946d65606f7c2f49766e4dddac5b4365
1e1ca8c4117b6b7271efc693f9685b4e907d8dc95324350392b21e94b3cffd18
```

可以通过 `docker ps` 命令来查看启动的 swarm manager 服务容器。

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
   NAMES
1e1ca8c4117b        swarm               "/swarm manage token:"
 11 seconds ago      Up 10 seconds       0.0.0.0:12375->2375/tcp
   jovial_rosalind
```

命令如果执行成功会返回刚启动的 Swarm 容器的 ID，此时一个简单的 Swarm 集群就已经搭建起来了，包括一个普通节点和一个管理节点。

查看集群节点列表

集群启动成功以后，用户可以在任何一台节点上使用 `swarm list` 命令查看集群中的节点列表。例如

```
$ docker run --rm swarm list token://946d65606f7c2f49766e4dddac5
b4365
192.168.0.2:2375
```

显示正是之前用 `swarm join` 命令加入集群的节点的地址。

我们在另外一台节点 `192.168.0.3` 上同样使用 `swarm join` 命令新加入一个节点：

```
$docker run --rm swarm join --addr=192.168.0.3:2375 token://946d
65606f7c2f49766e4dddac5b4365
time="2015-12-10T02:05:34Z" level=info msg="Registering on the d
iscovery service every 20s..." addr="192.168.0.3:2375" discovery=
"token://946d65606f7c2f49766e4dddac5b4365"
...
```

再次使用 `swarm list` 命令查看集群中的节点列表信息，可以看到新加入的节点：

```
$ docker run --rm swarm list token://946d65606f7c2f49766e4dddac5
b4365
192.168.0.3:2375
192.168.0.2:2375
```

使用集群服务

那么，怎么使用 Swarm 提供的服务呢？

实际上，所有 Docker 客户端可以继续使用，只要指定使用 Swarm manager 服务的监听地址即可。

例如，manager 服务监听的地址为 `192.168.0.2:12375`，则可以通过指定 `-H 192.168.0.2:12375` 选项来继续使用 Docker 客户端，执行任意 Docker 命令，例如 `ps`、`info`、`run` 等等。

在任意节点上使用 `docker run` 来启动若干容器，例如

```
$docker -H 192.168.0.2:12375:12375 run -d ubuntu ping 127.0.0.1
4c9bccbf86fb6e2243da58c1b15e9378fac362783a663426bbe7058eea84de46
```

使用 `ps` 命令查看集群中正在运行的容器。

```
$ docker -H 192.168.0.2:12375 ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
       NAMES
4c9bccbf86fb        ubuntu              "ping 127.0.0.1"
 About a minute ago   Up About a minute                       cl
ever_wright
730061a3801a        registry:latest     "docker-registry"
 2 minutes ago        Up 2 minutes         192.168.0.2:5000->500
0/tcp   Host-1/registry_registry_1
72d99f24a06f        redis:3.0           "/entrypoint.sh redis"
 2 minutes ago        Up 2 minutes         6379/tcp
         Host-1/registry_redis_1,Host-1/registry_registry_1/redi
s,Host-1/registry_registry_1/redis_1,Host-1/registry_registry_1/
registry_redis_1
```

输出结果中显示目前集群中正在运行的容器（注意不包括 Swarm manager 服务容器），可以在不同节点上使用 `docker ps` 查看本地容器，发现这些容器实际上可能运行在集群中多个节点上（被 Swarm 调度策略进行分配）。

使用 info 查看所有节点的信息。

```
$ docker -H 192.168.0.2:12375 info
Containers: 18
Images: 36
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 2
 Host-1: 192.168.0.2:2375
   └ Containers: 15
   └ Reserved CPUs: 0 / 4
   └ Reserved Memory: 1 GiB / 4.053 GiB
   └ Labels: executiondriver=native-0.2, kernelversion=3.16.0-43-
generic, operatingsystem=Ubuntu 14.04.3 LTS, storagedriver=aufs
 Host-2: 192.168.0.3:2375
   └ Containers: 3
   └ Reserved CPUs: 0 / 8
   └ Reserved Memory: 0 B / 16.46 GiB
   └ Labels: executiondriver=native-0.2, kernelversion=3.16.0-30-
generic, operatingsystem=Ubuntu 14.04.3 LTS, storagedriver=aufs
CPUs: 12
Total Memory: 20.51 GiB
Name: 1e1ca8c4117b
```

结果输出显示这个集群目前只有两个节点，地址分别是 `192.168.0.2` 和 `192.168.0.3` 。

类似的，也可以通过 Compose 模板来启动多个服务。不过请注意，要想让服务分布到多个 Swarm 节点上，需要采用版本 2 的写法。

使用网络

Swarm 为了支持跨主机的网络，默认采用了 `overlay` 网络类型，实现上通过 vxlan 来构建联通整个 Swarm 集群的网络。

首先，在集群中所有节点上，添加配置 Docker daemon 选项：

```
--cluster-store=<DISCOVERY_HOST:PORT> --cluster-advertise=<DOCKE
R_DAEMON_HOST:PORT>
```

以 consul 服务为例，可能类似：

```
--cluster-store=consul://<consul 服务地址>:8500 --cluster-adverti
se=192.168.0.3:2375
```

之后重启 Docker 服务。

首先，创建一个网络。

```
$ docker -H 192.168.0.2:12375 network create swarm_network
```

查看网络，将看到一个 overlay 类型的网络。

```
$ docker -H 192.168.0.2:12375 network ls
NETWORK ID          NAME                DRIVER
6edf2d16ec97        swarm_network       overlay
```

此时，所有添加到这个网络上的容器将自动被分配到集群中的节点上，并且彼此联通。

使用其它服务发现后端

Swarm 目前可以支持多种服务发现后端，这些后端功能上都是一致的，即维护属于某个集群的节点的信息。不同方案并无优劣之分，在实际使用时候，可以结合自身需求和环境限制进行选择，甚至自己定制其它方案。

使用中可以通过不同的路径来选择特定的服务发现后端机制。

- `token://<token>` ：使用 DockerHub 提供的服务，适用于可以访问公网情况；
- `file://path/to/file` ：使用本地文件，需要手动管理；
- `consul://<ip>/<path>` ：使用 consul 服务，私有环境推荐；
- `etcd://<ip1>,<ip2>/<path>` ：使用 etcd 服务，私有环境推荐；
- `zk://<ip1>,<ip2>/<path>` ：使用 zookeeper 服务，私有环境推荐；
- `[nodes://]<ip1>,<ip2>` ：手动指定集群中节点的地址，方便进行服务测试。

使用文件

使用本地文件的方式十分简单，就是讲所有属于某个集群的节点的 Docker daemon 信息写入一个文件中，然后让 manager 从这个文件中直接读取相关信息。

首先，在 Swarm 管理节点（ `192.168.0.2` ）上新建一个文件，把要加入集群的机器的 Docker daemon 信息写入文件：

```
$ tee /tmp/cluster_info <<-'EOF'
192.168.0.2:2375
192.168.0.3:2375
EOF
```

然后，本地执行 `swarm manage` 命令，并指定服务发现机制为本地文件，注意因为是容器方式运行 manager，需要将本地文件挂载到容器内。

```
$ docker run -d -p 12375:2375 -v /tmp/cluster_info:/tmp/cluster_
info swarm manage file:///tmp/cluster_info
```

接下来就可以通过使用 Swarm 服务来进行管理了，例如使用 info 查看所有节点的信息。

```
$ docker -H 192.168.0.2:12375 info
Containers: 18
Images: 36
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 2
 Host-1: 192.168.0.2:2375
  └ Containers: 15
  └ Reserved CPUs: 0 / 4
  └ Reserved Memory: 1 GiB / 4.053 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=3.16.0-43-
generic, operatingsystem=Ubuntu 14.04.3 LTS, storagedriver=aufs
 Host-2: 192.168.0.3:2375
  └ Containers: 3
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 16.46 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=3.16.0-30-
generic, operatingsystem=Ubuntu 14.04.3 LTS, storagedriver=aufs
CPUs: 12
Total Memory: 20.51 GiB
Name: e71eb5f1d48b
```

其它发现服务后端

其它服务发现后端的使用方法，也是大同小异，不同之处在于使用 Swarm 命令时指定的路径格式不同。

例如，对于前面介绍的 consul 服务后端来说。

快速部署一个 consul 服务的命令为：

```
$ docker run -d -p 8500:8500 --name=consul progrium/consul -serv
er -bootstrap
```

之后创建 Swarm 的管理服务，指定使用 consul 服务，管理端口监听在本地的 4000 端口。

```
$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication
 --advertise <manager_ip>:4000 consul://<consul_ip>:8500
```

Swarm 节点注册时候命令格式类似于：

```
$ swarm join --advertise=<node_ip:2375> consul://<consul_addr>/<
optional path prefix>
```

对于 etcd 服务后端来说，节点注册时候命令格式类似于：

```
$ swarm join --addr=<node_addr:2375> etcd://<etcd_addr1>,<etcd_a
ddr2>/<optional path prefix>
```

启动管理服务时候，格式类似于：

```
$ swarm manage -H tcp://<manager_ip>:4000 etcd://<etcd_addr1>,<e
tcd_addr2>/<optional path prefix>
```

地址和端口的范围匹配

对于基于文件，以及手动指定节点信息两种服务发现后端机制来说，其中地址和端口域可以支持指定一个范围，以一次性指定多个地址。 例如：

- `192.168.0.[2:10]:2375` 代表 `192.168.0.2:2375` -- `192.168.0.10:2375` 一共 9 个地址；
- `192.168.0.2:[2:9]375` 代表 `192.168.0.2:2375` -- `192.168.0.2:9375` 一共 8 个地址。

Swarm 中的调度器

调度是集群十分重要的功能，Swarm 目前支持三种调度策略：`spread`、`binpack` 和 `random`。

在执行`swarm manage`命令启动管理服务的时候，可以通过 `--strategy` 参数指定调度策略，默认的是 `spread`。

简单来说，这三种调度策略的优化目标如下：

- `spread`：如果节点配置相同，选择一个正在运行的容器数量最少的那个节点，即尽量平摊容器到各个节点；
- `binpack`：跟 `spread` 相反，尽可能的把所有的容器放在一台节点上面运行，即尽量少用节点，避免容器碎片化。
- `random`：直接随机分配，不考虑集群中节点的状态，方便进行测试使用。

spread 调度策略

仍然以之前创建好的集群为例，来演示下 spread 策略的行为。

在 `192.168.0.2` 节点启动管理服务，管理 token://946d65606f7c2f49766e4dddac5b4365 的集群。

```
$ docker run -d -p 12375:2375 swarm manage  --strategy "spread"
token://946d65606f7c2f49766e4dddac5b4365
c6f25e6e6abbe45c8bcf75ac674f2b64d5f31a5c6070d64ba954a0309b197930
```

列出集群中节点。

```
$ docker run --rm swarm list token://946d65606f7c2f49766e4dddac5
b4365
192.168.0.3:2375
192.168.0.2:2375
```

此时，两个节点上除了 swarm 外都没有运行其它容器。

启动一个 ubuntu 容器。

```
$ docker -H 192.168.0.2:12375 run -d ubuntu:14.04 ping 127.0.0.1
bac3dfda5306181140fc959969d738549d607bc598390f57bdd432d86f16f069
```

查看发现它实际上被调度到了 `192.168.0.3` 节点（当节点配置相同时候，初始节点随机选择）。

再次启动一个 ubuntu 容器。

```
$ docker -H 192.168.0.2:12375 run -d ubuntu:14.04 ping 127.0.0.1
8247067ba3a31e0cb692a8373405f95920a10389ce3c2a07091408281695281c
```

查看它的位置，发现被调度到了另外一个节点： `192.168.0.2` 节点。

```
$ docker -H 192.168.0.2:12375 ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
       NAMES
8247067ba3a3        ubuntu:14.04        "ping 127.0.0.1"
 1 minutes ago       Up 1 minutes                            Hos
t-2/sick_galileo
bac3dfda5306        ubuntu:14.04        "ping 127.0.0.1"
 2 minutes ago       Up 2 minutes                            Hos
t-3/compassionate_ritchie
```

当节点配置不同的时候， `spread` 会更愿意分配到配置较高的节点上。

binpack 调度策略

现在来看看 `binpack` 策略下的情况。

直接启动若干 ubuntu 容器，并查看它们的位置。

```
$ docker -H 192.168.0.2:12375 run -d ubuntu:14.04 ping 127.0.0.1
$ docker -H 192.168.0.2:12375 ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
       NAMES
4c4f45eba866        ubuntu:14.04        "ping 127.0.0.1"
 3 minutes ago       Up 3 minutes                            Hos
t-3/hopeful_brown
5e650541233c        ubuntu:14.04        "ping 127.0.0.1"
 3 minutes ago       Up 3 minutes                            Hos
t-3/pensive_wright
99c5a092530a        ubuntu:14.04        "ping 127.0.0.1"
 3 minutes ago       Up 3 minutes                            Hos
t-3/naughty_engelbart
4ab392c26eb2        ubuntu:14.04        "ping 127.0.0.1"
 3 minutes ago       Up 3 minutes                            Hos
t-3/thirsty_mclean
```

可以看到，所有的容器都是分布在同一个节点（`192.168.0.3`）上运行的。

Swarm 中的过滤器

Swarm 的调度器可以按照指定调度策略自动分配容器到节点。但有些时候希望能对这些分配加以干预。比如说，让 IO 敏感的容器分配到安装了 SSD 的节点上；让计算敏感的容器分配到 CPU 核数多的机器上；让网络敏感的容器分配到高带宽的机房；让某些容器尽量放同一个节点……。

这可以通过过滤器（filter）来实现，目前支持 `Constraint` 、 `Affinity` 、 `Port` 、 `Dependency` 、 `Health` 等五种过滤器。

Constraint 过滤器

Constraint 过滤器是绑定到节点的键值对，相当于给节点添加标签。

可在启动 Docker 服务的时候指定，例如指定某个节点颜色为 `red` 。

```
$ sudo docker daemon --label color=red -H tcp://0.0.0.0:2375 -H
unix:///var/run/docker.sock
```

同样的，可以写在 Docker 服务的配置文件里面（以 Ubuntu 14.04 为例，是 `/etc/default/docker` ）。

```
DOCKER_OPTS="--label color=red -H 0.0.0.0:2375 -H unix:///var/ru
n/docker.sock"
```

使用 Swarm 启动容器的时候，采用 `-e constarint:key=value` 的形式，可以过滤选择出匹配条件的节点。

例如，我们将 `192.168.0.2` 节点打上红色标签， `192.168.0.3` 节点打上绿色标签。

然后，分别启动两个容器，指定使用过滤器分别为红色和绿色。

```
$ docker -H 192.168.0.2:12375 run -d -e constraint:color==red ub
untu:14.04 ping 127.0.0.1
252ffb48e64e9858c72241f5eedf6a3e4571b1ad926faf091db3e26672370f64
$ docker -H 192.168.0.2:12375 run -d -e constraint:color==green
ubuntu:14.04 ping 127.0.0.1
3d6f8d7af8583416b17061d038545240c9e5c3be7067935d3ef2fbddce4b8136
```

注：指定标签中间是两个等号

查看它们将被分配到指定节点上。

```
$ docker -H 192.168.0.2:12375 ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
       NAMES
252ffb48e64e        ubuntu:14.04        "ping 127.0.0.1"
 1 minutes ago       Up 1 minutes                            Hos
t-2/sick_galileo
3d6f8d7af858        ubuntu:14.04        "ping 127.0.0.1"
 2 minutes ago       Up 2 minutes                            Hos
t-3/compassionate_ritchie
```

另外，Docker 内置了一些常见的过滤器，包括 `node` 、 `storagedriver` 、 `executiondriver` 、 `kernelversion` 、 `operatingsystem` 等。这些值可以通过 `docker info` 命令查看。

例如，目前集群中各个节点的信息为：

```
$ docker -H 192.168.0.2:12375 info
Containers: 5
Images: 39
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 2
 Host-2: 192.168.0.2:2375
  └ Containers: 4
  └ Reserved CPUs: 0 / 4
  └ Reserved Memory: 1 GiB / 4.053 GiB
  └ Labels: color=red, executiondriver=native-0.2, kernelversion
=3.16.0-43-generic, operatingsystem=Ubuntu 14.04.3 LTS, storaged
river=aufs
 Host-3: 192.168.0.3:2375
  └ Containers: 1
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 16.46 GiB
  └ Labels: color=green, executiondriver=native-0.2, kernelversi
on=3.16.0-30-generic, operatingsystem=Ubuntu 14.04.3 LTS, storag
edriver=aufs
CPUs: 12
Total Memory: 20.51 GiB
Name: 946d65606f7c
```

Affinity 过滤器

Affinity 过滤器允许用户在启动一个容器的时候，让它分配到某个已有容器的节点上。

例如，下面我们将启动一个 nginx 容器，让它分配到已经运行某个 ubuntu 容器的节点上。

在 Constraint 过滤器的示例中，我们分别启动了两个 ubuntu 容器 `sick_galileo` 和 `compassionate_ritchie`，分别在 Host-2 和 Host-3 上。

现在启动一个 nginx 容器，让它跟容器 `sick_galileo` 放在一起，都放到 Host-2 节点上。可以通过 `-e affinity:container==<name or id>` 参数来实现。

```
$ docker -H 192.168.0.2:12375 run -d -e affinity:container==sick
_galileo nginx
```

然后启动一个 redis 容器，让它跟容器 `compassionate_ritchie` 放在一起，都放到 Host-3 节点上。

```
$ docker -H 192.168.0.2:12375 run -d -e affinity:container==comp
assionate_ritchie redis
```

查看所有容器运行情况。

```
$ docker -H 192.168.0.2:12375 ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
       NAMES
0a32f15aa8ee        redis               "/entrypoint.sh redis"
 2 seconds ago       Up 1 seconds        6379/tcp
   Host-3/awesome_darwin
d2b9a53e67d5        nginx               "nginx -g 'daemon off"
 29 seconds ago      Up 28 seconds       80/tcp, 443/tcp
       Host-2/fervent_wilson
252ffb48e64e        ubuntu:14.04        "ping 127.0.0.1"
 2 minutes ago       Up 2 minutes                            Hos
t-2/sick_galileo
3d6f8d7af858        ubuntu:14.04        "ping 127.0.0.1"
 3 minutes ago       Up 3 minutes                            Hos
t-3/compassionate_ritchie
```

其它过滤器

其它过滤器的使用方法也是大同小异，例如通过 `-e affinity:image==<name or id>` 来选择拥有指定镜像的节点；通过 `-e affinity:label_name==value` 来选择拥有指定标签的容器所允许的节点。

此外，当容器端口需要映射到宿主机指定端口号的时候，Swarm 也会自动分配容器到指定宿主机端口可用的节点。

当不同容器之间存在数据卷或链接依赖的时候，Swarm 会分配这些容器到同一个节点上。

本章小结

本章笔者介绍了 Docker Swarm 的安装、使用和主要功能。

通过使用 Swarm，用户可以将若干 Docker 主机节点组成的集群当作一个大的虚拟 Docker 主机使用。并且，原先基于单机的 Docker 应用，可以无缝的迁移到 Swarm 上来。

实现这些功能的前提是服务自动发现能力。在现代分布式系统中，服务的自动发现、注册、更新等能力将成为系统的基本保障和重要基础。

在生产环境中，Swarm 的管理节点和发现服务后端要采用高可用性上的保护，可以采用集群模式。

值得一提的是，Swarm V2 功能已经被无缝嵌入到了 Docker 1.12+ 版本中，用户今后可以直接使用 Docker 命令来完成相关功能的配置，这将使得集群功能的管理更加简便。

etcd

etcd 是 CoreOS 团队发起的一个管理配置信息和服务发现（service discovery）的项目，在这一章里面，我们将介绍该项目的目标，安装和使用，以及实现的技术。

什么是 etcd



etcd 是 CoreOS 团队于 2013 年 6 月发起的开源项目，它的目标是构建一个高可用的分布式键值（key-value）数据库，基于 Go 语言实现。我们知道，在分布式系统中，各种服务的配置信息的管理分享，服务的发现是一个很基本同时也是很重要的问题。CoreOS 项目就希望基于 etcd 来解决这一问题。

etcd 目前在 github.com/coreos/etcd 进行维护，即将发布 2.0.0 版本。

受到 Apache ZooKeeper 项目和 doozer 项目的启发，etcd 在设计的时候重点考虑了下面四个要素：

- 简单：支持 REST 风格的 HTTP+JSON API
- 安全：支持 HTTPS 方式的访问
- 快速：支持并发 1k/s 的写操作
- 可靠：支持分布式结构，基于 Raft 的一致性算法

注：*Apache ZooKeeper* 是一套知名的分布式系统中进行同步和一致性管理的工具。 注：*doozer* 则是一个一致性分布式数据库。 注：*Raft* 是一套通过选举主节点来实现分布式系统一致性的算法，相比于大名鼎鼎的 *Paxos* 算法，它的过程更容易被人理解，由 *Stanford* 大学的 *Diego Ongaro* 和 *John Ousterhout* 提出。更多细节可以参考 *raftconsensus.github.io*。

一般情况下，用户使用 etcd 可以在多个节点上启动多个实例，并添加它们为一个集群。同一个集群中的 etcd 实例将会保持彼此信息的一致性。

安装

etcd 基于 Go 语言实现，因此，用户可以从 项目主页 下载源代码自行编译，也可以下载编译好的二进制文件，甚至直接使用制作好的 Docker 镜像文件来体验。

二进制文件方式下载

编译好的二进制文件都在 github.com/coreos/etcd/releases 页面，用户可以选择需要的版本，或通过下载工具下载。

例如，下面的命令使用 curl 工具下载压缩包，并解压。

```
curl -L  https://github.com/coreos/etcd/releases/download/v2.0.0
-rc.1/etcd-v2.0.0-rc.1-linux-amd64.tar.gz -o etcd-v2.0.0-rc.1-li
nux-amd64.tar.gz
tar xzvf etcd-v2.0.0-rc.1-linux-amd64.tar.gz
cd etcd-v2.0.0-rc.1-linux-amd64
```

解压后，可以看到文件包括

```
$ ls
etcd  etcdctl  etcd-migrate  README-etcdctl.md  README.md
```

其中 etcd 是服务主文件，etcdctl 是提供给用户的命令客户端，etcd-migrate 负责进行迁移。

推荐通过下面的命令将三个文件都放到系统可执行目录 `/usr/local/bin/` 或 `/usr/bin/` 。

```
$ sudo cp etcd* /usr/local/bin/
```

运行 etcd，将默认组建一个两个节点的集群。数据库服务端默认监听在 2379 和 4001 端口，etcd 实例监听在 2380 和 7001 端口。显示类似如下的信息：

```
$ ./etcd
```

```
2014/12/31 14:52:09 no data-dir provided, using default data-dir
 ./default.etcd
2014/12/31 14:52:09 etcd: listening for peers on http://localhos
t:2380
2014/12/31 14:52:09 etcd: listening for peers on http://localhos
t:7001
2014/12/31 14:52:09 etcd: listening for client requests on http:
//localhost:2379
2014/12/31 14:52:09 etcd: listening for client requests on http:
//localhost:4001
2014/12/31 14:52:09 etcdserver: name = default
2014/12/31 14:52:09 etcdserver: data dir = default.etcd
2014/12/31 14:52:09 etcdserver: snapshot count = 10000
2014/12/31 14:52:09 etcdserver: advertise client URLs = http://l
ocalhost:2379,http://localhost:4001
2014/12/31 14:52:09 etcdserver: initial advertise peer URLs = ht
tp://localhost:2380,http://localhost:7001
2014/12/31 14:52:09 etcdserver: initial cluster = default=http:/
/localhost:2380,default=http://localhost:7001
2014/12/31 14:52:10 etcdserver: start member ce2a822cea30bfca in
 cluster 7e27652122e8b2ae
2014/12/31 14:52:10 raft: ce2a822cea30bfca became follower at te
rm 0
2014/12/31 14:52:10 raft: newRaft ce2a822cea30bfca [peers: [], t
erm: 0, commit: 0, lastindex: 0, lastterm: 0]
2014/12/31 14:52:10 raft: ce2a822cea30bfca became follower at te
rm 1
2014/12/31 14:52:10 etcdserver: added local member ce2a822cea30b
fca [http://localhost:2380 http://localhost:7001] to cluster 7e2
7652122e8b2ae
2014/12/31 14:52:11 raft: ce2a822cea30bfca is starting a new ele
ction at term 1
2014/12/31 14:52:11 raft: ce2a822cea30bfca became candidate at t
erm 2
2014/12/31 14:52:11 raft: ce2a822cea30bfca received vote from ce
2a822cea30bfca at term 2
2014/12/31 14:52:11 raft: ce2a822cea30bfca became leader at term
 2
2014/12/31 14:52:11 raft.node: ce2a822cea30bfca elected leader c
e2a822cea30bfca at term 2
```

```
2014/12/31 14:52:11 etcdserver: published {Name:default ClientUR
Ls:[http://localhost:2379 http://localhost:4001]} to cluster 7e2
7652122e8b2ae
```

此时，可以使用 etcdctl 命令进行测试，设置和获取键值 `testkey: "hello world"` ，检查 etcd 服务是否启动成功：

```
$ ./etcdctl set testkey "hello world"
hello world
$ ./etcdctl get testkey
hello world
```

说明 etcd 服务已经成功启动了。

当然，也可以通过 HTTP 访问本地 2379 或 4001 端口的方式来进行操作，例如查看 `testkey` 的值：

```
$ curl -L http://localhost:4001/v2/keys/testkey
{"action":"get","node":{"key":"/testkey","value":"hello world","
modifiedIndex":3,"createdIndex":3}}
```

Docker 镜像方式下载

镜像名称为 quay.io/coreos/etcd:v2.0.0_rc.1，可以通过下面的命令启动 etcd 服务监听到 4001 端口。

```
$ sudo docker run -p 4001:4001 -v /etc/ssl/certs/:/etc/ssl/certs
/ quay.io/coreos/etcd:v2.0.0_rc.1
```

使用 etcdctl

etcdctl 是一个命令行客户端，它能提供一些简洁的命令，供用户直接跟 etcd 服务打交道，而无需基于 HTTP API 方式。这在某些情况下将很方便，例如用户对服务进行测试或者手动修改数据库内容。我们也推荐在刚接触 etcd 时通过 etcdctl 命令来熟悉相关的操作，这些操作跟 HTTP API 实际上是对应的。

etcd 项目二进制发行包中已经包含了 etcdctl 工具，没有的话，可以从 github.com/coreos/etcd/releases 下载。

etcdctl 支持如下的命令，大体上分为数据库操作和非数据库操作两类，后面将分别进行解释。

```
$ etcdctl -h
NAME:
   etcdctl - A simple command line client for etcd.

USAGE:
   etcdctl [global options] command [command options] [arguments
...]

VERSION:
   2.0.0-rc.1

COMMANDS:
   backup    backup an etcd directory
   mk         make a new key with a given value
   mkdir     make a new directory
   rm         remove a key
   rmdir     removes the key if it is an empty directory or a key
-value pair
   get         retrieve the value of a key
   ls         retrieve a directory
   set         set the value of a key
   setdir    create a new or existing directory
   update    update an existing key with a given value
   updatedir    update an existing directory
   watch    watch a key for changes
```

```
    exec-watch    watch a key for changes and exec an executable
    member    member add, remove and list subcommands
    help, h    Shows a list of commands or help for one command

GLOBAL OPTIONS:
    --debug            output cURL commands which can be used to
reproduce the request
    --no-sync            don't synchronize cluster information be
fore sending request
    --output, -o 'simple'    output response in the given format
(`simple` or `json`)
    --peers, -C             a comma-delimited list of machine add
resses in the cluster (default: "127.0.0.1:4001")
    --cert-file             identify HTTPS client using this SSL
certificate file
    --key-file             identify HTTPS client using this SSL k
ey file
    --ca-file             verify certificates of HTTPS-enabled se
rvers using this CA bundle
    --help, -h            show help
    --version, -v        print the version
```

数据库操作

数据库操作围绕对键值和目录的 CRUD （符合 REST 风格的一套操作：Create）完整生命周期的管理。

etcd 在键的组织上采用了层次化的空间结构（类似于文件系统中目录的概念），用户指定的键可以为单独的名字，如 `testkey` ，此时实际上放在根目录 `/` 下面，也可以为指定目录结构，如 `cluster1/node2/testkey` ，则将创建相应的目录结构。

注：*CRUD* 即 *Create, Read, Update, Delete*，是符合 *REST* 风格的一套 *API* 操作。

set

指定某个键的值。例如

```
$ etcdctl set /testdir/testkey "Hello world"
Hello world
```

支持的选项包括：

```
--ttl '0'            该键值的超时时间（单位为秒），不配置（默认为 0）则永不超时
--swap-with-value value 若该键现在的值是 value，则进行设置操作
--swap-with-index '0'    若该键现在的索引值是指定索引，则进行设置操作
```

get

获取指定键的值。例如

```
$ etcdctl set testkey hello
hello
$ etcdctl update testkey world
world
```

当键不存在时，则会报错。例如

```
$ etcdctl get testkey2
Error:  100: Key not found (/testkey2) [1]
```

支持的选项为

```
--sort    对结果进行排序
--consistent 将请求发给主节点，保证获取内容的一致性
```

update

当键存在时，更新值内容。例如

```
$ etcdctl set testkey hello
hello
$ etcdctl update testkey world
world
```

当键不存在时，则会报错。例如

```
$ etcdctl update testkey2 world
Error:  100: Key not found (/testkey2) [1]
```

支持的选项为

```
--ttl '0'    超时时间（单位为秒），不配置（默认为 0）则永不超时
```

rm

删除某个键值。例如

```
$ etcdctl rm testkey
```

当键不存在时，则会报错。例如

```
$ etcdctl rm testkey2
Error:  100: Key not found (/testkey2) [8]
```

支持的选项为

```
--dir          如果键是个空目录或者键值对则删除
--recursive          删除目录和所有子键
--with-value      检查现有的值是否匹配
--with-index '0'     检查现有的 index 是否匹配
```

mk

如果给定的键不存在，则创建一个新的键值。例如

```
$ etcdctl mk /testdir/testkey "Hello world"
Hello world
```

当键存在的时候，执行该命令会报错，例如

```
$ etcdctl set testkey "Hello world"
Hello world
$ ./etcdctl mk testkey "Hello world"
Error:  105: Key already exists (/testkey) [2]
```

支持的选项为

```
--ttl '0'    超时时间（单位为秒），不配置（默认为 0）则永不超时
```

mkdir

如果给定的键目录不存在，则创建一个新的键目录。例如

```
$ etcdctl mkdir testdir
```

当键目录存在的时候，执行该命令会报错，例如

```
$ etcdctl mkdir testdir
$ etcdctl mkdir testdir
Error:  105: Key already exists (/testdir) [7]
```

支持的选项为

```
--ttl '0'    超时时间（单位为秒），不配置（默认为 0）则永不超时
```

setdir

创建一个键目录，无论存在与否。

支持的选项为

```
--ttl '0'    超时时间（单位为秒），不配置（默认为 0）则永不超时
```

updatedir

更新一个已经存在的目录。 支持的选项为

```
--ttl '0'    超时时间（单位为秒），不配置（默认为 0）则永不超时
```

rmdir

删除一个空目录，或者键值对。

若目录不空，会报错

```
$ etcdctl set /dir/testkey hi
hi
$ etcdctl rmdir /dir
Error:  108: Directory not empty (/dir) [13]
```

ls

列出目录（默认为根目录）下的键或者子目录，默认不显示子目录中内容。

例如

```
$ ./etcdctl set testkey 'hi'
hi
$ ./etcdctl set dir/test 'hello'
hello
$ ./etcdctl ls
/testkey
/dir
$ ./etcdctl ls dir
/dir/test
```

支持的选项包括

```
--sort    将输出结果排序
--recursive    如果目录下有子目录，则递归输出其中的内容
-p        对于输出为目录，在最后添加 `/` 进行区分
```

非数据库操作

backup

备份 etcd 的数据。

支持的选项包括

```
--data-dir          etcd 的数据目录
--backup-dir      备份到指定路径
```

watch

监测一个键值的变化，一旦键值发生更新，就会输出最新的值并退出。

例如，用户更新 testkey 键值为 Hello world。

```
$ etcdctl watch testkey
Hello world
```

支持的选项包括

```
--forever          一直监测，直到用户按 `CTRL+C` 退出
--after-index '0'     在指定 index 之前一直监测
--recursive         返回所有的键值和子键值
```

exec-watch

监测一个键值的变化，一旦键值发生更新，就执行给定命令。

例如，用户更新 testkey 键值。

```
$etcdctl exec-watch testkey -- sh -c 'ls'
default.etcd
Documentation
etcd
etcdctl
etcd-migrate
README-etcdctl.md
README.md
```

支持的选项包括

```
--after-index '0'     在指定 index 之前一直监测
--recursive          返回所有的键值和子键值
```

member

通过 list、add、remove 命令列出、添加、删除 etcd 实例到 etcd 集群中。

例如本地启动一个 etcd 服务实例后，可以用如下命令进行查看。

```
$ etcdctl member list
ce2a822cea30bfca: name=default peerURLs=http://localhost:2380,ht
tp://localhost:7001 clientURLs=http://localhost:2379,http://loca
lhost:4001
```

命令选项

- `--debug` 输出 cURL 命令，显示执行命令的时候发起的请求
- `--no-sync` 发出请求之前不同步集群信息
- `--output, -o 'simple'` 输出内容的格式 (`simple` 为原始信息，`json` 为进行json格式解码，易读性好一些)
- `--peers, -C` 指定集群中的同伴信息，用逗号隔开 (默认为: "127.0.0.1:4001")
- `--cert-file` HTTPS 下客户端使用的 SSL 证书文件
- `--key-file` HTTPS 下客户端使用的 SSL 密钥文件
- `--ca-file` 服务端使用 HTTPS 时，使用 CA 文件进行验证

- `--help, -h` 显示帮助命令信息
- `--version, -v` 打印版本信息

CoreOS

CoreOS的设计是为你提供能够像谷歌一样的大型互联网公司一样的基础设施管理能力来动态扩展和管理的计算能力。

CoreOS的安装文件和运行依赖非常小,它提供了精简的Linux系统。它使用Linux容器在更高的抽象层来管理你的服务，而不是通过常规的YUM和APT来安装包。

同时，CoreOS几乎可以运行在任何平台：Vagrant, Amazon EC2, QEMU/KVM, VMware 和 OpenStack 等等，甚至你所使用的硬件环境。

CoreOS介绍

提起Docker，我们不得不提的就是CoreOS.

CoreOS对Docker甚至容器技术的发展都带来了巨大的推动作用。

CoreOS是一种支持大规模服务部署的Linux系统。

CoreOS使得在基于最小化的现代操作系统上构建规模化的计算仓库成为了可能。

CoreOS是一个新的Linux发行版。通过重构，CoreOS提供了运行现代基础设施的特性。

CoreOS的这些策略和架构允许其它公司像Google，Facebook和Twitter那样高弹性的运行自己得服务。

CoreOS遵循Apache 2.0协议并且可以运行在现有的硬件或云提供商之上。

CoreOS特性

一个最小化操作系统

CoreOS被设计成一个来构建你平台的最小化的现代操作系统。

它比现有的Linux安装平均节省40%的RAM（大约114M）并允许从 PXE/iPXE 非常快速的启动。

无痛更新

利用主动和被动双分区方案来更新OS，使用分区作为一个单元而不是一个包一个包得更新。

这使得每次更新变得快速，可靠，而且很容易回滚。

Docker容器

应用作为Docker容器运行在CoreOS上。容器以包得形式提供最大得灵活性并且可以在几毫秒启动。

支持集群

CoreOS可以在一个机器上很好地运行，但是它被设计用来搭建集群。

可以通过fleet很容易得使应用容器部署在多台机器上并且通过服务发现把他们连接在一起。

分布式系统工具

内置诸如分布式锁和主选举等原生工具用来构建大规模分布式系统得构建模块。

服务发现

很容易定位服务在集群的那里运行并当发生变化时进行通知。它是复杂高动态集群必不可少的。在CoreOS中构建高可用和自动故障负载。

CoreOS工具介绍

CoreOS提供了三大工具，它们分别是：服务发现，容器管理和进程管理。

使用etcd服务发现

CoreOS的第一个重要组件就是使用etcd来实现的服务发现。

如果你使用默认的样例cloud-config文件，那么etcd会在启动时自动运行。

例如：

```
#cloud-config

hostname: coreos0
ssh_authorized_keys:
  - ssh-rsa AAAA...
coreos:
  units:
    - name: etcd.service
      command: start
    - name: fleet.service
      command: start
  etcd:
    name: coreos0
    discovery: https://discovery.etcd.io/<token>
```

配置文件里有一个token，获取它可以通过如下方式：

访问地址

https://discovery.etcd.io/new

你将会获取一个包含你得teoken得URL。

通过Docker进行容器管理

第二个组件就是docker，它用来运行你的代码和应用。

每一个CoreOS的机器上都安装了它，具体使用请参考本书其他章节。

使用fleet进行进程管理

第三个CoreOS组件是fleet。

它是集群的分布式初始化系统。你应该使用fleet来管理你的docker容器的生命周期。

Fleet通过接受systemd单元文件来工作，同时在你集群的机器上通过单元文件中编写的偏好来对它们进行调度。

首先，让我们构建一个简单的可以运行docker容器的systemd单元。把这个文件保存在home目录并命名为hello.service：

```
hello.service

[Unit]
Description=My Service
After=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill hello
ExecStartPre=-/usr/bin/docker rm hello
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "w
hile true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop hello
```

然后，读取并启动这个单元：

```
$ fleetctl load hello.service
=> Unit hello.service loaded on 8145ebb7.../172.17.8.105
$ fleetctl start hello.service
=> Unit hello.service launched on 8145ebb7.../172.17.8.105
```

这样，你的容器将在集群里被启动。

下面我们查看下它的状态：

```
$ fleetctl status hello.service
● hello.service - My Service
   Loaded: loaded (/run/fleet/units/hello.service; linked-runtim
e)
   Active: active (running) since Wed 2014-06-04 19:04:13 UTC; 4
4s ago
 Main PID: 27503 (bash)
   CGroup: /system.slice/hello.service
           ├─27503 /bin/bash -c /usr/bin/docker start -a hello |
| /usr/bin/docker run --name hello busybox /bin/sh -c "while tru
e; do echo Hello World; sleep 1; done"
           └─27509 /usr/bin/docker run --name hello busybox /bin
/sh -c while true; do echo Hello World; sleep 1; done

Jun 04 19:04:57 core-01 bash[27503]: Hello World
..snip...
Jun 04 19:05:06 core-01 bash[27503]: Hello World
```

我们可以停止容器：

```
fleetctl destroy hello.service
```

至此，就是CoreOS提供的三大工具。

快速搭建CoreOS集群

在这里我们要搭建一个集群环境，毕竟单机环境没有什么挑战不是？

然后为了在你的电脑运行一个集群环境，我们使用Vagrant。

*Vagrant*的使用这里不再阐述，请自行学习

如果你第一次接触CoreOS这样的分布式平台，运行一个集群看起来好像一个很复杂的任务，这里我们给你展示在本地快速搭建一个CoreOS集群环境是多么的容易。

准备工作

首先要确认在你本地的机器上已经安装了最新版本的Virtualbox, Vagrant 和 git。

这是我们可以在本地模拟集群环境的前提条件，如果你已经拥有，请继续，否则自行搜索学习。

配置工作

从CoreOS官方代码库获取基本配置，并进行修改

首先，获取模板配置文件

```
git clone https://github.com/coreos/coreos-vagrant
cd coreos-vagrant
cp user-data.sample user-data
```

获取新的token

```
curl https://discovery.etcd.io/new
```

把获取的token放到user-data文件中，示例如下：

```
#cloud-config

coreos:
  etcd:
    discovery: https://discovery.etcd.io/<token>
```

启动集群

默认情况下，CoreOS Vagrantfile 将会启动单机。

我们需要复制并修改config.rb.sample文件.

复制文件

```
cp config.rb.sample config.rb
```

修改集群配置参数num_instances为3。

启动集群

```
vagrant up
=>
Bringing machine 'core-01' up with 'virtualbox' provider...
Bringing machine 'core-02' up with 'virtualbox' provider...
Bringing machine 'core-03' up with 'virtualbox' provider...
==> core-01: Box 'coreos-alpha' could not be found. Attempting t
o find and install...
    core-01: Box Provider: virtualbox
    core-01: Box Version: >= 0
==> core-01: Adding box 'coreos-alpha' (v0) for provider: virtua
lbox
    core-01: Downloading: http://storage.core-os.net/coreos/amd6
4-usr/alpha/coreos_production_vagrant.box
    core-01: Progress: 46% (Rate: 6105k/s, Estimated time remain
ing: 0:00:16)
```

添加ssh的公匙

```
ssh-add ~/.vagrant.d/insecure_private_key
```

连接集群中的第一台机器

```
vagrant ssh core-01 -- -A
```

测试集群

使用fleet来查看机器运行状况

```
fleetctl list-machines
=>
MACHINE     IP              METADATA
517d1c7d... 172.17.8.101  -
cb35b356... 172.17.8.103  -
17040743... 172.17.8.102  -
```

如果你也看到了如上类似的信息，恭喜，本地基于三台机器的集群已经成功启动，是不是很简单。

那么之后你就可以基于CoreOS的三大工具做任务分发，分布式存储等很多功能了。

Kubernetes

Kubernetes 是 Google 团队发起并维护的基于Docker的开源容器集群管理系统，它不仅支持常见的云平台，而且支持内部数据中心。

建于 Docker 之上的 Kubernetes 可以构建一个容器的调度服务，其目的是让用户透过Kubernetes集群来进行云端容器集群的管理，而无需用户进行复杂的设置工作。系统会自动选取合适的工作节点来执行具体的容器集群调度处理工作。其核心概念是Container Pod（容器仓）。一个Pod是有一组工作于同一物理工作节点的容器构成的。这些组容器拥有相同的网络命名空间/IP以及存储配额，可以根据实际情况对每一个Pod进行端口映射。此外，Kubernetes工作节点会由主系统进行管理，节点包含了能够运行Docker容器所用到的服务。

本章将分为 5 节介绍 Kubernetes。包括

- 项目简介
- 快速入门
- 基本概念
- 实践例子
- 架构分析等高级话题

项目简介



Kubernetes 是 Google 团队发起的开源项目，它的目标是管理跨多个主机的容器，提供基本的部署，维护以及运用伸缩，主要实现语言为Go语言。Kubernetes是：

- 易学：轻量级，简单，容易理解
- 便携：支持公有云，私有云，混合云，以及多种云平台
- 可拓展：模块化，可插拔，支持钩子，可任意组合
- 自修复：自动重调度，自动重启，自动复制

Kubernetes构建于Google数十年经验，一大半来源于Google生产环境规模的经验。结合了社区最佳的想法和实践。

在分布式系统中，部署，调度，伸缩一直是最为重要的也最为基础的功能。Kubernets就是希望解决这一序列问题的。

Kubernets 目前在github.com/GoogleCloudPlatform/kubernetes进行维护，截至定稿最新版本为 0.7.2 版本。

Kubernetes 能够运行在任何地方！

虽然Kubernets最初是为GCE定制的，但是在后续版本中陆续增加了其他云平台的支持，以及本地数据中心的支持。

快速上手

目前，Kubenetes 支持在多种环境下的安装，包括本地主机（Fedora）、云服务（Google GAE、AWS 等）。然而最快速体验 Kubernetes 的方式显然是本地通过 Docker 的方式来启动相关进程。

下图展示了在单节点使用 Docker 快速部署一套 Kubernetes 的拓扑。

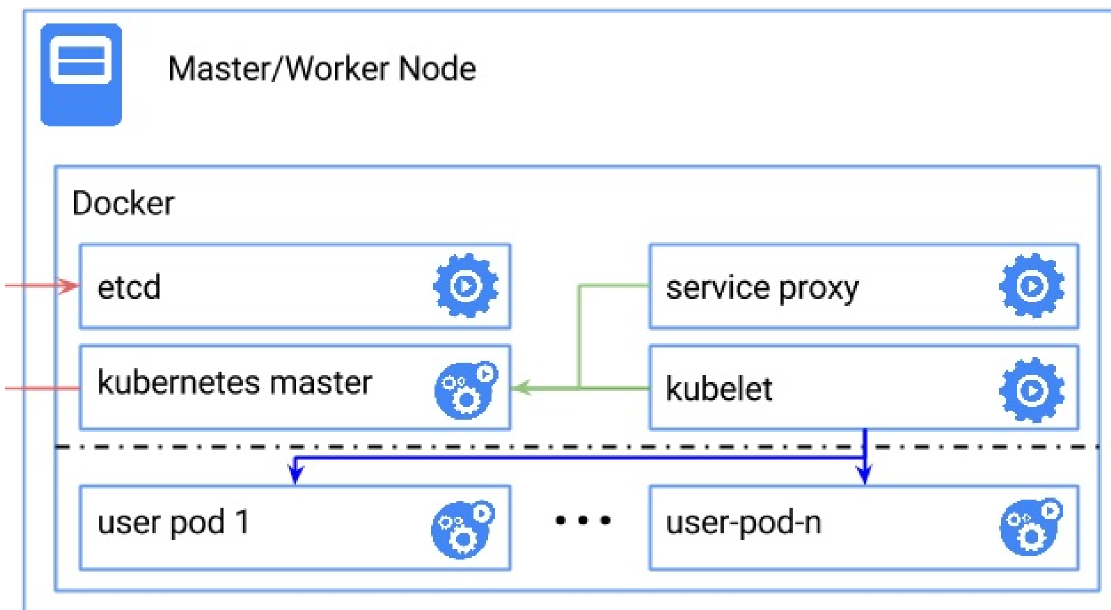


图 *1.20.2.1 - 在 Docker 中启动 Kubernetes*

Kubernetes 依赖 Etcd 服务来维护所有主节点的状态。

启动 **Etcd** 服务。

```
docker run --net=host -d gcr.io/google_containers/etcd:2.0.9 /usr/local/bin/etcd --addr=127.0.0.1:4001 --bind-addr=0.0.0.0:4001 --data-dir=/var/etcd/data
```

启动主节点

启动 kubelet。

```
docker run --net=host -d -v /var/run/docker.sock:/var/run/docker
.sock  gcr.io/google_containers/hyperkube:v0.17.0 /hyperkube kub
elet --api_servers=http://localhost:8080 --v=2 --address=0.0.0.0
 --enable_server --hostname_override=127.0.0.1 --config=/etc/kub
ernetes/manifests
```

启动服务代理

```
docker run -d --net=host --privileged gcr.io/google_containers/h
yperkube:v0.17.0 /hyperkube proxy --master=http://127.0.0.1:8080
 --v=2
```

测试状态

在本地访问 8080 端口，应该获取到类似如下的结果：

```
$ curl 127.0.0.1:8080
{
  "paths": [
    "/api",
    "/api/v1beta1",
    "/api/v1beta2",
    "/api/v1beta3",
    "/healthz",
    "/healthz/ping",
    "/logs/",
    "/metrics",
    "/static/",
    "/swagger-ui/",
    "/swaggerapi/",
    "/validate",
    "/version"
  ]
}
```

查看服务

所有服务启动后过一会，查看本地实际运行的 Docker 容器，应该有如下几个。

```
CONTAINER ID        IMAGE
 COMMAND                CREATED             STATUS
PORTS               NAMES
ee054db2516c        gcr.io/google_containers/hyperkube:v0.17.0
"/hyperkube schedule   2 days ago          Up 1 days
                   k8s_scheduler.509f29c9_k8s-master-127.0.0.1_d
efault_9941e5170b4365bd4aa91f122ba0c061_e97037f5
3b0f28de07a2        gcr.io/google_containers/hyperkube:v0.17.0
 "/hyperkube apiserve   2 days ago          Up 1 days
                    k8s_apiserver.245e44fa_k8s-master-127.0.0.1_
default_9941e5170b4365bd4aa91f122ba0c061_6ab5c23d
2eaa44ecdd8e        gcr.io/google_containers/hyperkube:v0.17.0
"/hyperkube controll   2 days ago          Up 1 days
                   k8s_controller-manager.33f83d43_k8s-master-12
7.0.0.1_default_9941e5170b4365bd4aa91f122ba0c061_1a60106f
30aa7163cbef        gcr.io/google_containers/hyperkube:v0.17.0
 "/hyperkube proxy --   2 days ago          Up 1 days
                    jolly_davinci
a2f282976d91        gcr.io/google_containers/pause:0.8.0
"/pause"               2 days ago          Up 2 days
                   k8s_POD.e4cc795_k8s-master-127.0.0.1_default_
9941e5170b4365bd4aa91f122ba0c061_e8085b1f
c060c52acc36        gcr.io/google_containers/hyperkube:v0.17.0
"/hyperkube kubelet    2 days ago          Up 1 days
                   serene_nobel
cc3cd263c581        gcr.io/google_containers/etcd:2.0.9
 "/usr/local/bin/etcd   2 days ago          Up 1 days
                    happy_turing
```

这些服务大概分为三类：主节点服务、工作节点服务和其它服务。

主节点服务

- apiserver 是整个系统的对外接口，提供 RESTful 方式供客户端和其它组件调

用；

- scheduler 负责对资源进行调度，分配某个 pod 到某个节点上；
- controller-manager 负责管理控制器，包括 endpoint-controller（刷新服务和 pod 的关联信息）和 replication-controller（维护某个 pod 的复制为配置的数值）。

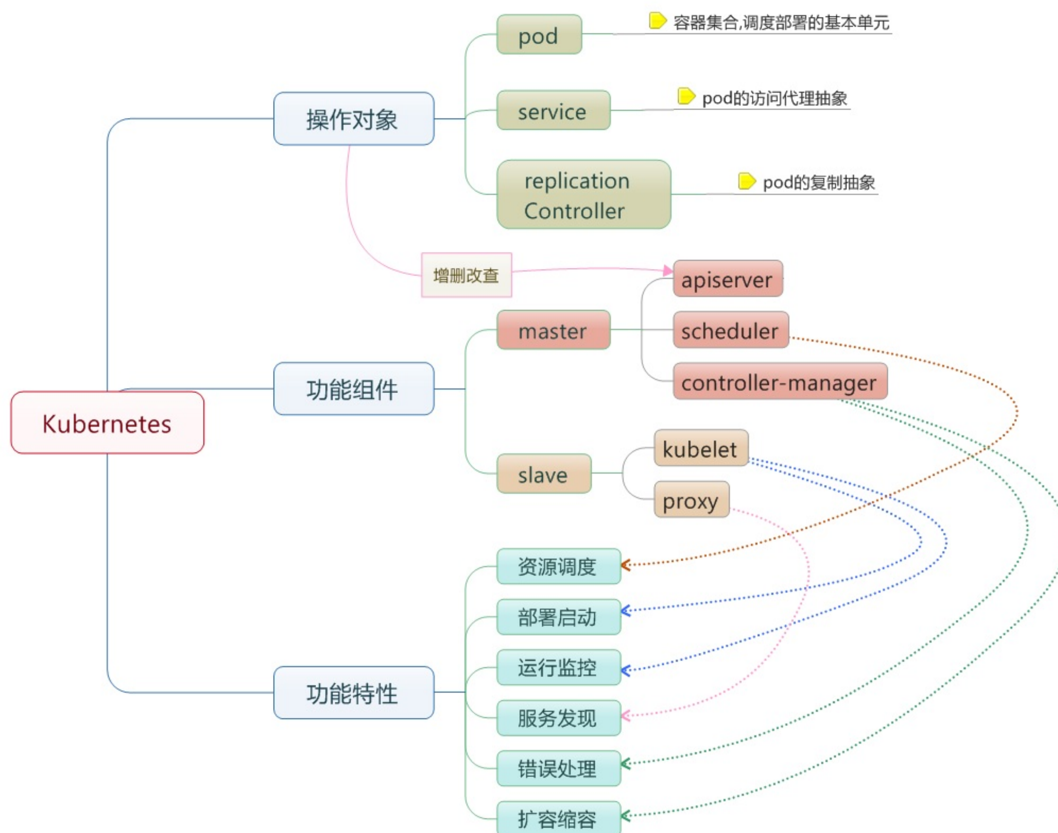
工作节点服务

- kubelet 是工作节点执行操作的 agent，负责具体的容器生命周期管理，根据从数据库中获取的信息来管理容器，并上报 pod 运行状态等；
- proxy 为 pod 上的服务提供访问的代理。

其它服务

- etcd 是所有状态的存储数据库；
- `gcr.io/google_containers/pause:0.8.0` 是 Kubernetes 启动后自动 pull 下来的测试镜像。

基本概念



- 节点（Node）：一个节点是一个运行 Kubernetes 中的主机。
- 容器组（Pod）：一个 Pod 对应于由若干容器组成的一个容器组，同个组内的容器共享一个存储卷(volume)。
- 容器组生命周期（pos-states）：包含所有容器状态集合，包括容器组状态类型，容器组生命周期，事件，重启策略，以及replication controllers。
- Replication Controllers（replication-controllers）：主要负责指定数量的pod在同一时间一起运行。
- 服务（services）：一个Kubernetes服务是容器组逻辑的高级抽象，同时也对外提供访问容器组的策略。
- 卷（volumes）：一个卷就是一个目录，容器对其有访问权限。
- 标签（labels）：标签是用来连接一组对象的，比如容器组。标签可以被用来组织和选择子对象。
- 接口权限（accessing_the_api）：端口，ip地址和代理的防火墙规则。
- web 界面（ux）：用户可以通过 web 界面操作Kubernetes。
- 命令行操作（cli）：`kubecfg` 命令。

节点

在 Kubernetes 中，节点是实际工作的点，以前叫做 Minion。节点可以是虚拟机或者物理机器，依赖于一个集群环境。每个节点都有一些必要的服务以运行容器组，并且它们都可以通过主节点来管理。必要服务包括 Docker，kubelet 和代理服务。

容器状态

容器状态用来描述节点的当前状态。现在，其中包含三个信息：

主机IP

主机IP需要云平台来查询，Kubernetes把它作为状态的一部分来保存。如果Kubernetes没有运行在云平台上，节点ID就是必需的。IP地址可以变化，并且可以包含多种类型的IP地址，如公共IP，私有IP，动态IP，ipv6等等。

节点周期

通常来说节点有 `Pending` ， `Running` ， `Terminated` 三个周期，如果Kubernetes发现了一个节点并且其可用，那么Kubernetes就把它标记为 `Pending` 。然后在某个时刻，Kubernetes将会标记其为 `Running` 。节点的结束周期称为 `Terminated` 。一个已经terminated的节点不会接受和调度任何请求，并且已经在其上运行的容器组也会删除。

节点状态

节点的状态主要是用来描述处于 `Running` 的节点。当前可用的有 `NodeReachable` 和 `NodeReady` 。以后可能会增加其他状态。 `NodeReachable` 表示集群可达。 `NodeReady` 表示kubelet返回 StatusOk并且HTTP状态检查健康。

节点管理

节点并非Kubernetes创建，而是由云平台创建，或者就是物理机器、虚拟机。在Kubernetes中，节点仅仅是一条记录，节点创建之后，Kubernetes会检查其是否可用。在Kubernetes中，节点用如下结构保存：

```
{
  "id": "10.1.2.3",
  "kind": "Minion",
  "apiVersion": "v1beta1",
  "resources": {
    "capacity": {
      "cpu": 1000,
      "memory": 1073741824
    },
  },
  "labels": {
    "name": "my-first-k8s-node",
  },
}
```

Kubernetes校验节点可用依赖于id。在当前的版本中，有两个接口可以用来管理节点：节点控制和Kube管理。

节点控制

在Kubernetes主节点中，节点控制器是用来管理节点的组件。主要包含：

- 集群范围内节点同步
- 单节点生命周期管理

节点控制有一个同步轮寻，主要监听所有云平台的虚拟实例，会根据节点状态创建和删除。可以通过 `--node_sync_period` 标志来控制该轮寻。如果一个实例已经创建，节点控制将会为其创建一个结构。同样的，如果一个节点被删除，节点控制也会删除该结构。在Kubernetes启动时可用通过 `--machines` 标记来显示指定节点。同样可以使用 `kubectl` 来一条一条的添加节点，两者是相同的。通过设置 `--sync_nodes=false` 标记来禁止集群之间的节点同步，你也可以使用api/kubectl 命令行来增删节点。

容器组

在Kubernetes中，使用的最小单位是容器组，容器组是创建，调度，管理的最小单位。 一个容器组使用相同的Dokcer容器并共享卷（挂载点）。一个容器组是一个特定运用的打包集合，包含一个或多个容器。

和运行的容器类似，一个容器组被认为只有很短的运行周期。容器组被调度到一组节点运行，知道容器的生命周期结束或者其被删除。如果节点死掉，运行在其上的容器组将会被删除而不是重新调度。（也许在将来的版本中会添加容器组的移动）。

容器组设计的初衷

资源共享和通信

容器组主要是为了数据共享和它们之间的通信。

在一个容器组中，容器都使用相同的网络地址和端口，可以通过本地网络来相互通信。每个容器组都有独立的ip，可用通过网络来和其他物理主机或者容器通信。

容器组有一组存储卷（挂载点），主要是为了让容器在重启之后可以不丢失数据。

容器组管理

容器组是一个运用管理和部署的高层次抽象，同时也是一组容器的接口。容器组是部署、水平放缩的最小单位。

容器组的使用

容器组可以通过组合来构建复杂的运用，其本来的意义包含：

- 内容管理，文件和数据加载以及本地缓存管理等。
- 日志和检查点备份，压缩，快照等。
- 监听数据变化，跟踪日志，日志和监控代理，消息发布等。
- 代理，网桥
- 控制器，管理，配置以及更新

替代方案

为什么不在一个单一的容器里运行多个程序？

- 1.透明化。为了使容器组中的容器保持一致的基础设施和服务，比如进程管理和资源监控。这样设计是为了用户的便利性。
- 2.解偶软件之间的依赖。每个容器都可能重新构建和发布，Kubernetes必须支持热发布和热更新（将来）。
- 3.方便使用。用户不必运行独立的程序管理，也不用担心每个运用程序的退出状态。
- 4.高效。考虑到基础设施有更多的职责，容器必须要轻量化。

容器组的生命状态

包括若干状态值：pending、running、succeeded、failed。

pending

容器组已经被节点接受，但有一个或多个容器还没有运行起来。这将包含某些节点正在下载镜像的时间，这种情形会依赖于网络情况。

running

容器组已经被调度到节点，并且所有的容器都已经启动。至少有一个容器处于运行状态（或者处于重启状态）。

succeeded

所有的容器都正常退出。

failed

容器组中所有容器都意外中断了。

容器组生命周期

通常来说，如果容器组被创建了就不会自动销毁，除非被某种行为触发，而触发此种情况可能是人为，或者复制控制器所为。唯一例外的是容器组由 succeeded状态成功退出，或者在一定时间内重试多次依然失败。

如果某个节点死掉或者不能连接，那么节点控制器将会标记其上的容器组的状态为 `failed` 。

举例如下。

- 容器组状态 `running` ，有 1 容器，容器正常退出
    - 记录完成事件
    - 如果重启策略为：
        - 始终：重启容器，容器组保持 `running`
        - 失败时：容器组变为 `succeeded`
        - 从不：容器组变为 `succeeded`
- 容器组状态 `running` ，有1容器，容器异常退出
    - 记录失败事件
    - 如果重启策略为：
        - 始终：重启容器，容器组保持 `running`
        - 失败时：重启容器，容器组保持 `running`
        - 从不：容器组变为 `failed`
- 容器组状态 `running` ，有2容器，有1容器异常退出
    - 记录失败事件
    - 如果重启策略为：
        - 始终：重启容器，容器组保持 `running`
        - 失败时：重启容器，容器组保持 `running`
        - 从不：容器组保持 `running`
            - 当有2容器退出
        - 记录失败事件
        - 如果重启策略为：
            - 始终：重启容器，容器组保持 `running`
            - 失败时：重启容器，容器组保持 `running`
            - 从不：容器组变为 `failed`
- 容器组状态 `running` ，容器内存不足
    - 标记容器错误中断
    - 记录内存不足事件
    - 如果重启策略为：
        - 始终：重启容器，容器组保持 `running`
        - 失败时：重启容器，容器组保持 `running`
        - 从不：记录错误事件，容器组变为 `failed`
- 容器组状态 `running` ，一块磁盘死掉

    - 杀死所有容器
    - 记录事件
    - 容器组变为 `failed`
    - 如果容器组运行在一个控制器下，容器组将会在其他地方重新创建
- 容器组状态 `running` ，对应的节点段溢出
    - 节点控制器等到超时
    - 节点控制器标记容器组 `failed`
    - 如果容器组运行在一个控制器下，容器组将会在其他地方重新创建

Replication Controllers

服务

卷

标签

接口权限

web界面

命令行操作

kubectl 使用

kubectl 是 Kubernetes 自带的客户端，可以用它来直接操作 Kubernetes。

使用格式有两种：

```
kubectl [flags]
kubectl [command]
```

get

Display one or many resources

describe

Show details of a specific resource

create

Create a resource by filename or stdin

update

Update a resource by filename or stdin.

delete

Delete a resource by filename, stdin, resource and ID, or by resources and label selector.

namespace

SUPERCEDED: Set and view the current Kubernetes namespace

log

Print the logs for a container in a pod.

rolling-update

Perform a rolling update of the given ReplicationController.

resize

Set a new size for a Replication Controller.

exec

Execute a command in a container.

port-forward

Forward one or more local ports to a pod.

proxy

Run a proxy to the Kubernetes API server

run-container

Run a particular image on the cluster.

stop

Gracefully shut down a resource by id or filename.

expose

Take a replicated application and expose it as Kubernetes Service

label

Update the labels on a resource

config

config modifies kubeconfig files

cluster-info

Display cluster info

api-versions

Print available API versions.

version

Print the client and server version information.

help

Help about any command

基本架构

任何优秀的项目都离不开优秀的架构设计。本小节将介绍 Kubernetes 在架构方面的设计考虑。

基本考虑

如果让我们自己从头设计一套容器管理平台，有如下几个方面是很容易想到的：

- 分布式架构，保证扩展性；
- 逻辑集中式的控制平面 + 物理分布式的运行平面；
- 一套资源调度系统，管理哪个容器该分配到哪个节点上；
- 一套对容器内服务进行抽象和 HA 的系统。

运行原理

下面这张图完整展示了 Kubernetes 的运行原理。

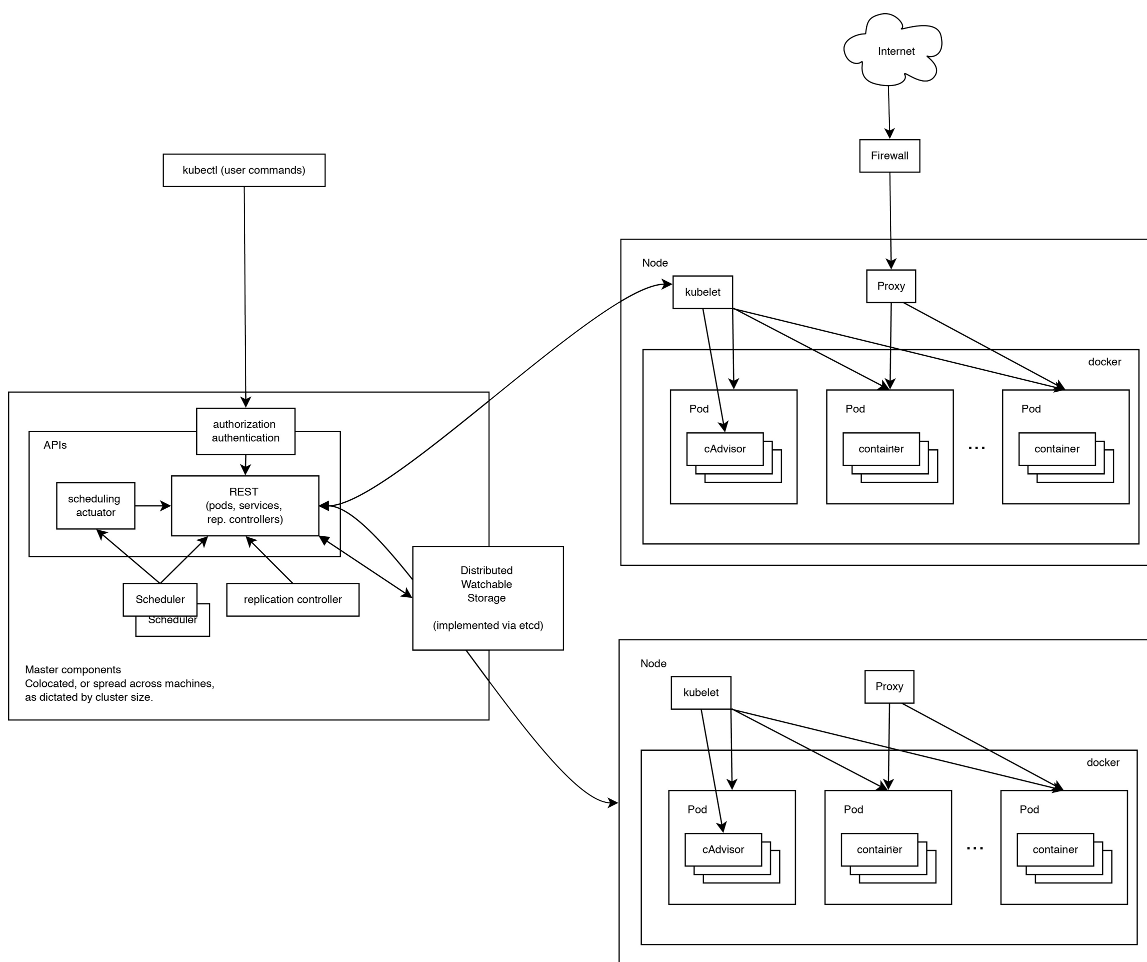


图 *1.20.5.1 - Kubernetes* 架构

可见，Kubernetes 首先是一套分布式系统，由多个节点组成，节点分为两类：一类是属于管理平面的主节点/控制节点（Master Node）；一类是属于运行平面的工作节点（Worker Node）。

显然，复杂的工作肯定都交给控制节点去做了，工作节点负责提供稳定的操作接口和能力抽象即可。

从这张图上，我们没有能发现 Kubernetes 中对于控制平面的分布式实现，但是由于数据后端自身就是一套分布式的数据库（Etcd），因此可以很容易扩展到分布式实现。

控制平面

主节点服务

主节点上需要提供如下的管理服务：

- apiserver 是整个系统的对外接口，提供一套 RESTful 的 Kubernetes API，供客户端和其它组件调用；
- scheduler 负责对资源进行调度，分配某个 pod 到某个节点上。是 pluggable 的，意味着很容易选择其它实现方式；
- controller-manager 负责管理控制器，包括 endpoint-controller（刷新服务和 pod 的关联信息）和 replication-controller（维护某个 pod 的复制为配置的数值）。

Etcd

这里 Etcd 即作为数据后端，又作为消息中间件。

通过 Etcd 来存储所有的主节点上的状态信息，很容易实现主节点的分布式扩展。

组件可以自动的去侦测 Etcd 中的数值变化来获得通知，并且获得更新后的数据来执行相应的操作。

工作节点

- kubelet 是工作节点执行操作的 agent，负责具体的容器生命周期管理，根据从数据库中获取的信息来管理容器，并上报 pod 运行状态等；
- kube-proxy 是一个简单的网络访问代理，同时也是一个 Load Balancer。它负责将访问到某个服务的请求具体分配给工作节点上的 Pod（同一类标签）。

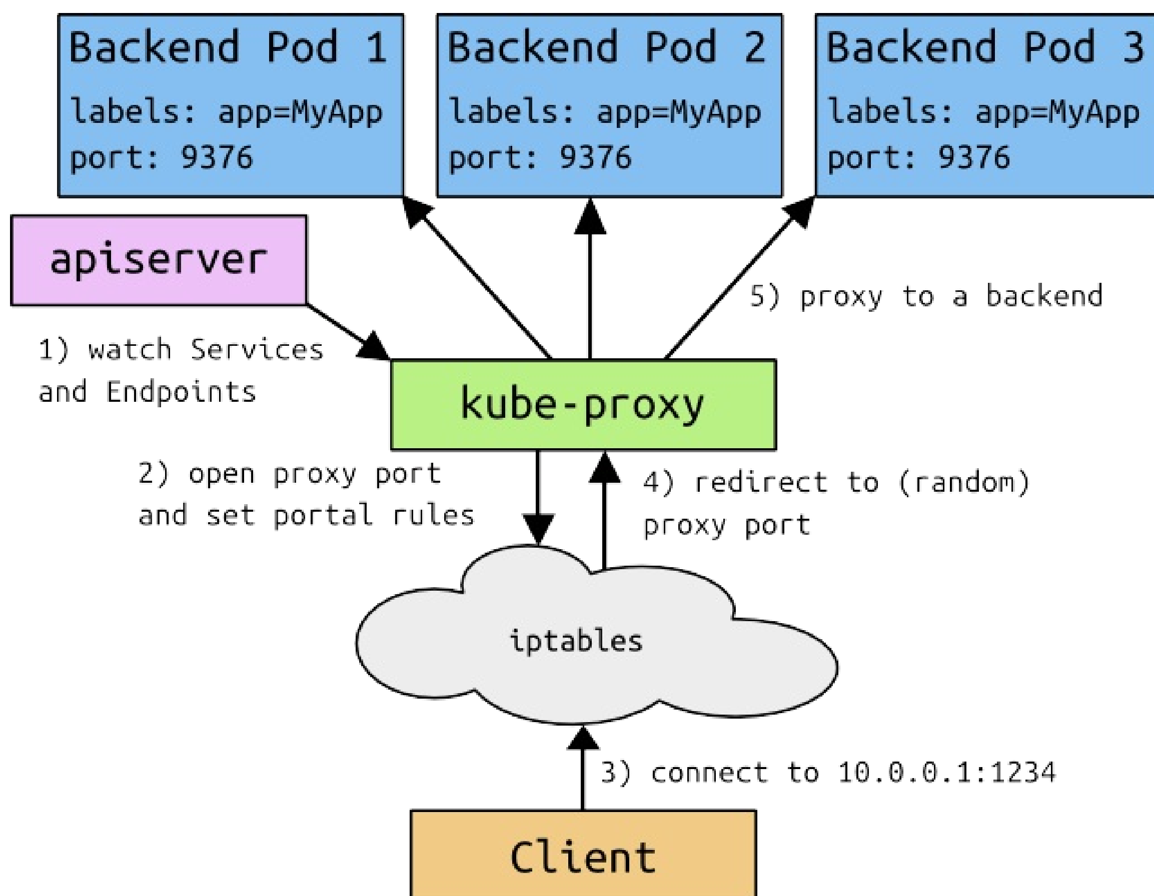


图 *1.20.5.2 - Proxy* 代理对服务的请求

Mesos - 优秀的集群资源调度平台

Mesos 项目是源自 UC Berkeley 的对集群资源进行抽象和管理的开源项目，类似于操作系统内核，用户可以使用它很容易地实现分布式应用的自动化调度。

同时，Mesos 自身也很好地结合和主持了 Docker 等相关容器技术，基于 Mesos 已有的大量应用框架，可以实现用户应用的快速上线。

本章将介绍 Mesos 项目的安装、使用、配置以及核心的原理知识。

简介

Mesos 最初由 UC Berkeley 的 AMP 实验室于 2009 年发起，遵循 Apache 协议，目前已经成立了 Mesosphere 公司进行运营。Mesos 可以将整个数据中心的资源（包括 CPU、内存、存储、网络等）进行抽象和调度，使得多个应用同时运行在集群中分享资源，并无需关心资源的物理分布情况。

如果把数据中心中的集群资源看做一台服务器，那么 Mesos 要做的事情，其实就是今天操作系统内核的职责：抽象资源 + 调度任务。Mesos 项目是 Mesosphere 公司 Datacenter Operating System (DCOS) 产品的核心部件。

Mesos 项目主要由 C++ 语言编写，项目官方地址为 http://mesos.apache.org，代码仍在快速演化中，已经发布了正式版 1.0.0 版本。

Mesos 拥有许多引人注目的特性，包括：

- 支持数万个节点的大规模场景（Apple、Twitter、eBay 等公司实践）；
- 支持多种应用框架，包括 Marathon、Singularity、Aurora 等；
- 支持 HA（基于 ZooKeeper 实现）；
- 支持 Docker、LXC 等容器机制进行任务隔离；
- 提供了多个流行语言的 API，包括 Python、Java、C++ 等；
- 自带了简洁易用的 WebUI，方便用户直接进行操作。

值得注意的是，Mesos 自身只是一个资源抽象的平台，要使用它往往需要结合运行其上的分布式应用（在 Mesos 中被称作框架，framework），比如 Hadoop、Spark 等可以进行分布式计算的大数据处理应用；比如 Marathon 可以实现 PaaS，快速部署应用并自动保持运行；比如 ElasticSearch 可以索引海量数据，提供灵活的整合和查询能力......

大部分时候，用户只需要跟这些框架打交道即可，完全无需关心底下的资源调度情况，因为 Mesos 已经自动帮你实现了。这大大方便了上层应用的开发和运维。

当然，用户也可以基于 Mesos 打造自己的分布式应用框架。

Mesos 安装与使用

以 Mesos 结合 Marathon 应用框架为例，来看下如何快速搭建一套 Mesos 平台。

Marathon 是可以跟 Mesos 一起协作的一个 framework，基于 Scala 实现，可以实现保持应用的持续运行。

另外，Mesos 默认利用 ZooKeeper 来进行多个主节点之间的选举，以及从节点发现主节点的过程。一般在生产环境中，需要启动多个 Mesos master 服务（推荐 3 或 5 个），并且推荐使用 supervisord 等进程管理器来自动保持服务的运行。

ZooKeeper 是一个分布式集群中信息同步的工具，通过自动在多个节点中选举 `leader`，保障多个节点之间的某些信息保持一致性。

安装

安装主要需要 mesos、zookeeper 和 marathon 三个软件包。

Mesos 也采用了经典的主-从结构，一般包括若干主节点和大量从节点。其中，mesos master 服务和 zookeeper 需要部署到所有的主节点，mesos slave 服务需要部署到所有从节点。marathon 可以部署到主节点。

安装可以通过源码编译、软件源或者 Docker 镜像方式进行，下面分别进行介绍。

源码编译

源码编译方式可以保障获取到最新版本，但编译过程比较费时间。

首先，从 apache.org 开源网站下载最新的源码。

```
$ git clone https://git-wip-us.apache.org/repos/asf/mesos.git
```

其中，主要代码在 `src` 目录下，应用框架代码在 `frameworks` 目录下，文档在 `docs` 目录下，`include` 中包括了跟 Mesos 打交道使用的一些 API 定义头文件。

安装依赖，主要包括 Java 运行环境、Linux 上的自动编译环境等。

```
$ sudo apt-get update
$ sudo apt-get install -y openjdk-8-jdk autoconf libtool \
build-essential python-dev python-boto libcurl4-nss-dev \
libsasl2-dev maven libapr1-dev libsvn-dev
```

后面就是常规 C++ 项目的方法，configure 之后利用 Makefile 进行编译和安装。

```
$ cd mesos
$ ./bootstrap
$ mkdir build
$ cd build && ../configure --with-network-isolator
$ make
$ make check && sudo make install
```

软件源安装

通过软件源方式进行安装相对会省时间，但往往不是最新版本。

这里以 Ubuntu 系统为例，首先添加软件源地址。

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E5615
1BF
$ DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
$ CODENAME=$(lsb_release -cs)
$ echo "deb http://repos.mesosphere.io/${DISTRO} ${CODENAME} mai
n" | \
sudo tee /etc/apt/sources.list.d/mesosphere.list
```

刷新本地软件仓库信息并安装 zookeeper、mesos、marathon 三个软件包。

```
$ sudo apt-get -y update && sudo apt-get -y install zookeeper me
sos marathon
```

注意，Marathon 最新版本需要 jdk 1.8+ 的支持。如果系统中有多个 Java 版本，需要检查配置默认的 JDK 版本符合要求。

```
$ sudo update-alternatives --config java
```

安装 Mesos 成功后，会在 `/usr/sbin/` 下面发现 `mesos-master` 和 `mesos-slave` 两个二进制文件，分别对应主节点上需要运行的管理服务和从节点上需要运行的任务服务。

用户可以手动运行二进制文件启动服务，也可以通过 `service` 命令来方便进行管理。

例如，在主节点上重启 Mesos 管理服务：

```
$ sudo service mesos-master restart
```

通过 `service` 命令来管理，实际上是通过调用 `/usr/bin/mesos-init-wrapper` 脚本文件进行处理。

基于 Docker

需要如下三个镜像。

- ZooKeeper：https://registry.hub.docker.com/u/garland/zookeeper/
- Mesos：https://registry.hub.docker.com/u/garland/mesosphere-docker-mesos-master/
- Marathon：https://registry.hub.docker.com/u/garland/mesosphere-docker-marathon/

其中 mesos-master 镜像在后面将分别作为 master 和 slave 角色进行使用。

首先，拉取三个镜像。

```
$ docker pull garland/zookeeper
$ docker pull garland/mesosphere-docker-mesos-master
$ docker pull garland/mesosphere-docker-marathon
```

导出主节点机器的地址到环境变量。

```
$ HOST_IP=10.0.0.2
```

在主节点上启动 Zookeepr 容器。

```
docker run -d \
-p 2181:2181 \
-p 2888:2888 \
-p 3888:3888 \
garland/zookeeper
```

在主节点上启动 Mesos Master 服务容器。

```
docker run --net="host" \
-p 5050:5050 \
-e "MESOS_HOSTNAME=${HOST_IP}" \
-e "MESOS_IP=${HOST_IP}" \
-e "MESOS_ZK=zk://${HOST_IP}:2181/mesos" \
-e "MESOS_PORT=5050" \
-e "MESOS_LOG_DIR=/var/log/mesos" \
-e "MESOS_QUORUM=1" \
-e "MESOS_REGISTRY=in_memory" \
-e "MESOS_WORK_DIR=/var/lib/mesos" \
-d \
garland/mesosphere-docker-mesos-master
```

在主节点上启动 Marathon。

```
docker run \
-d \
-p 8080:8080 \
garland/mesosphere-docker-marathon --master zk://${HOST_IP}:2181
/mesos --zk zk://${HOST_IP}:2181/marathon
```

在从节点上启动 Mesos slave 容器。

```
docker run -d \
--name mesos_slave_1 \
--entrypoint="mesos-slave" \
-e "MESOS_MASTER=zk://${HOST_IP}:2181/mesos" \
-e "MESOS_LOG_DIR=/var/log/mesos" \
-e "MESOS_LOGGING_LEVEL=INFO" \
garland/mesosphere-docker-mesos-master:latest
```

接下来，可以通过访问本地 8080 端口来使用 Marathon 启动任务了。

配置说明

下面以本地通过软件源方式安装为例，解释如何修改各个配置文件。

ZooKeepr

ZooKeepr 是一个分布式应用的协调工具，用来管理多个主节点的选举和冗余，监听在 2181 端口。推荐至少布置三个主节点来被 ZooKeeper 维护。

配置文件默认都在 `/etc/zookeeper/conf/` 目录下。比较关键的配置文件有两个： `myid` 和 `zoo.cfg` 。

myid 文件会记录加入 ZooKeeper 集群的节点的序号（1-255之间）。 `/var/lib/zookeeper/myid` 文件其实也是软连接到了该文件。

比如配置某节点序号为 1，则需要在该节点上执行：

```
$ echo 1 | sudo dd of=/etc/zookeeper/conf/myid
```

节点序号在 ZooKeeper 集群中必须唯一，不能出现多个拥有相同序号的节点。

另外，需要修改 zoo.cfg 文件，该文件是主配置文件，主要需要添加上加入 ZooKeeper 集群的机器的序号和对应监听地址。

例如，现在 ZooKeeper 集群中有三个节点，地址分别为 `10.0.0.2` 、 `10.0.0.3` 、 `10.0.0.4` ，序号分别配置为 `2` 、 `3` 、 `4` 。

则配置如下的三行：

```
server.2=10.0.0.2:2888:3888
server.3=10.0.0.3:2888:3888
server.4=10.0.0.4:2888:3888
```

其中第一个端口 2888 负责从节点连接到主节点的；第二个端口 3888 则负责主节点进行选举时候通信。

也可以用主机名形式，则需要各个节点 `/etc/hosts` 文件中都记录地址到主机名对应的映射关系。

完成配置后，启动 ZooKeeper 服务。

```
$ sudo service zookeeper start
```

Mesos

Mesos 的默认配置目录有三个：

- /etc/mesos/：主节点和从节点都会读取的配置文件，最关键的是 zk 文件存放主节点的信息；
- /etc/mesos-master/：只有主节点会读取的配置，等价于启动 mesos-master 命令时候的默认选项；
- /etc/mesos-slave/：只有从节点会读取的配置，等价于启动 mesos-master 命令时候的默认选项。

最关键的是需要在所有节点上修改 `/etc/mesos/zk` ，写入主节点集群的 ZooKeeper 地址列表，例如：

```
zk://10.0.0.2:2181,10.0.0.3:2181,10.0.0.4:2181/mesos
```

此外， `/etc/default/mesos` 、 `/etc/default/mesos-master` 、 `/etc/default/mesos-slave` 这三个文件中可以存放一些环境变量定义，Mesos 服务启动之前，会将这些环境变量导入进来作为启动参数。格式为 `MESOS_OPTION_NAME` 。

下面分别说明在主节点和从节点上的配置。

主节点

一般只需要关注 `/etc/mesos-master/` 目录下的文件。默认情况下目录下为空。

该目录下文件命名和内容需要跟 mesos-master 支持的命令行选项一一对应。可以通过 `mesos-master --help` 命令查看支持的选项。

例如某个文件 `key` 中内容为 `value` ，则在 mesos-master 服务启动的时候，会自动添加参数 `--key=value` 给二进制命令。

例如，mesos-master 服务默认监听在 loopback 端口，即 `127.0.0.1:5050` ，我们需要修改主节点监听的地址，则可以创建 /etc/mesos-master/ip 文件，在其中写入主节点监听的外部地址。

为了正常启动 mesos-master 服务，还需要指定 `work_dir` 参数(表示应用框架的工作目录)的值，可以通过创建 /etc/mesos-master/work_dir 文件，在其中写入目录，例如 `/var/lib/mesos` 。工作目录下会生成一个 `replicated_log` 目录，会存有各种同步状态的持久化信息。

以及指定 quorum 参数的值，该参数用来表示 ZooKeeper 集群中要求最少参加表决的节点数目。一般设置为比 ZooKeeper 集群中节点个数的半数多一些（比如三个节点的话，可以配置为 `2` ）。

此外，要修改 Mesos 集群的名称，可以创建 `/etc/mesos-master/cluster` 文件，在其中写入集群的别名，例如 `MesosCluster` 。

总结下，建议在 `/etc/mesos-master` 目录下，配置至少四个参数文件： `ip` 、 `quorum` 、 `work_dir` 、 `cluster` 。

修改配置之后，需要启动服务即可生效。

```
$ sudo service mesos-master start
```

更多选项可以参考后面的配置项解析章节。

主节点服务启动后，则可以在从节点上启动 mesos-slave 服务来加入主节点的管理。

从节点

一般只需要关注 `/etc/mesos-slave/` 目录下的文件。默认情况下目录下为空。

文件命名和内容也是跟主节点类似，对应二进制文件支持的命令行参数。

建议在从节点上，创建 `/etc/mesos-slave/ip` 文件，在其中写入跟主节点通信的地址。

修改配置之后，也需要重新启动服务。

```
$ sudo service mesos-slave start
```

更多选项可以参考后面的配置项解析章节。

Marathon

Marathon 作为 Mesos 的一个应用框架，配置要更为简单，必需的配置项有 `--master` 和 `--zk` 。

安装完成后，会在 /usr/bin 下多一个 marathon shell 脚本，为启动 marathon 时候执行的命令。

配置目录为 `/etc/marathon/conf` （需要手动创建），此外默认配置文件在 `/etc/default/marathon` 。

我们手动创建配置目录，并添加配置项（文件命名和内容跟 Mesos 风格一致），让 Marathon 能连接到已创建的 Mesos 集群中。

```
$ sudo mkdir -p /etc/marathon/conf
$ sudo cp /etc/mesos/zk /etc/marathon/conf/master
```

同时，让 Marathon 也将自身的状态信息保存到 ZooKeeper 中。创建 `/etc/marathon/conf/zk` 文件，添加 ZooKeeper 地址和路径。

```
zk://10.0.0.2:2181,10.0.0.2:2181,10.0.0.2:2181/marathon
```

启动 marathon 服务。

```
$ sudo service marathon start
```

访问 Mesos 图形界面

Mesos 自带了 Web 图形界面，可以方便用户查看集群状态。

用户在 Mesos 主节点服务和从节点服务都启动后，可以通过浏览器访问主节点 5050 端口，看到类似如下界面，已经有两个 slave 节点加入了。

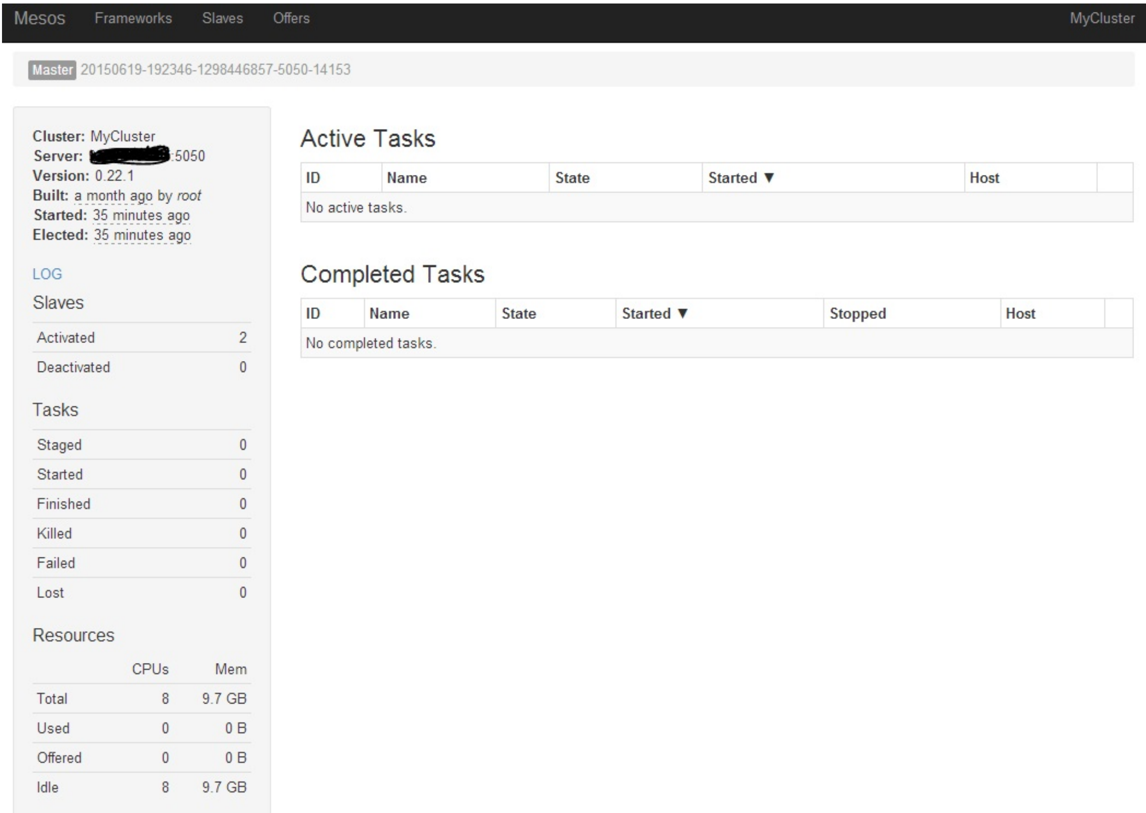


图 *1.21.2.1 - mesos* 界面查看加入的 *slave* 节点

通过 Slaves 标签页能看到加入集群的从节点的信息。

如果没有启动 Marathon 服务，在 Frameworks 标签页下将看不到任何内容。

访问 **Marathon** 图形界面

Marathon 服务启动成功后，在 Mesos 的 web 界面的 Frameworks 标签页下面将能看到名称为 marathon 的框架出现。

同时可以通过浏览器访问 8080 端口，看到 Marathon 自己的管理界面。

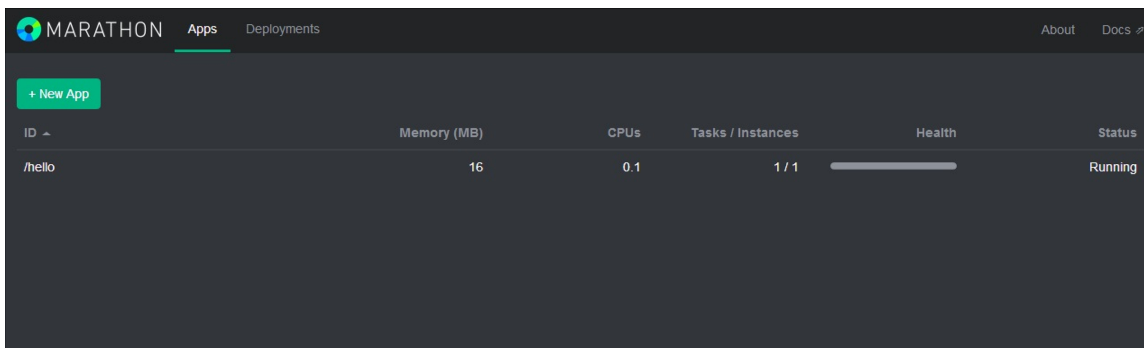


图 *1.21.2.2 - marathon* 图形管理界面

此时，可以通过界面或者 REST API 来创建一个应用，Marathon 会保持该应用的持续运行。

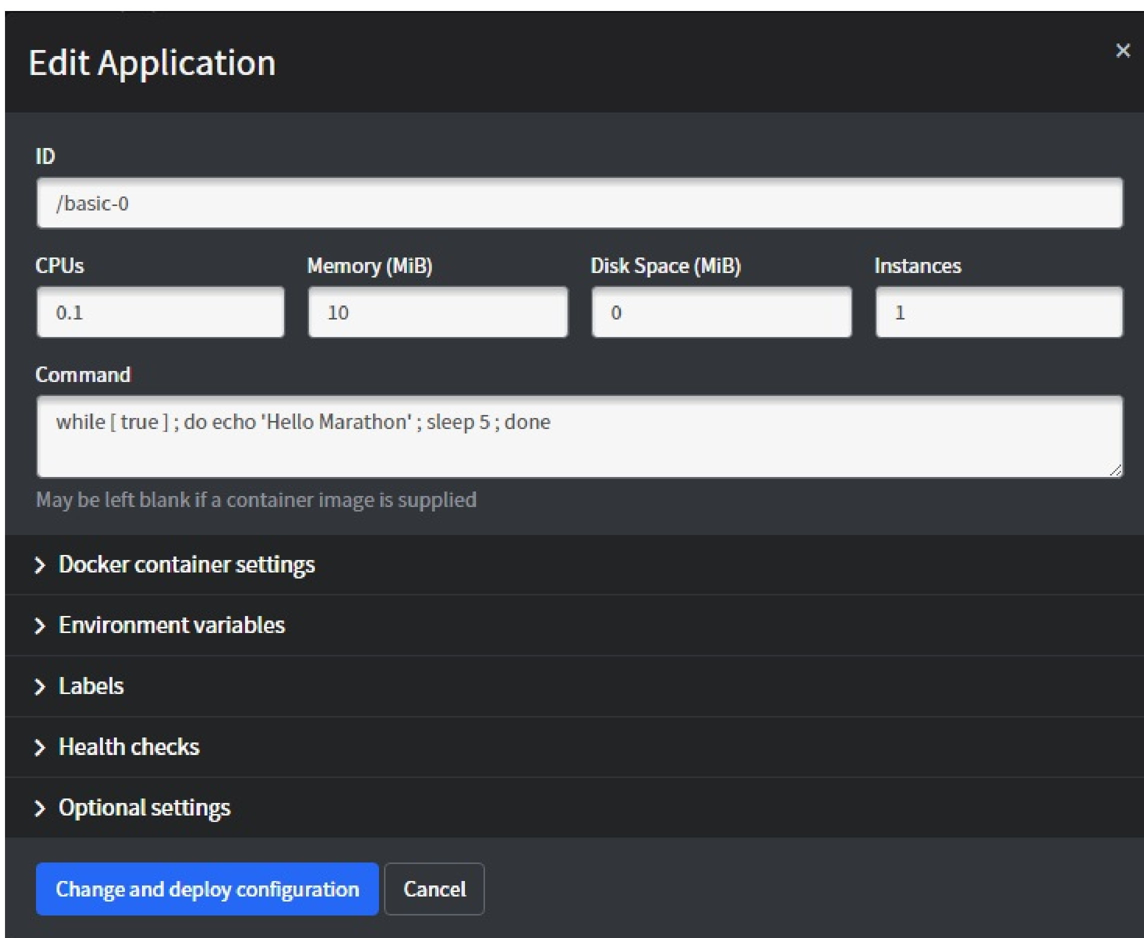


图 *1.21.2.3 - marathon* 查看任务支持的参数

通过界面方式可以看到各任务支持的参数（包括资源、命令、环境变量、健康检查等），同时可以很容易地修改任务运行实例数进行扩展，非常适合进行测试。

如果要更自动化地使用 Marathon，则需要通过它的 REST API 进行操作。

一般的，启动新任务需要先创建一个定义模板（JSON 格式），然后发到指定的 API。

例如，示例任务 basic-0 的定义模板为：

```
{
    "id": "basic-0",
    "cmd": "while [ true ] ; do echo 'Hello Marathon' ; sleep 5
; done",
    "cpus": 0.1,
    "mem": 10.0,
    "instances": 1
}
```

该任务申请资源为 0.1 个单核 CPU 资源和 10 MB 的内存资源，具体命令为每隔五秒钟用 shell 打印一句 `Hello Marathon` 。

可以通过如下命令发出 basic-0 任务到 Marathon 框架，框架会分配任务到某个满足条件的从节点上，成功会返回一个 json 对象，描述任务的详细信息。

```
$ curl -X POST http://marathon_host:8080/v2/apps -d @basic-0.jso
n -H "Content-type: application/json"
{"id":"/basic-0","cmd":"while [ true ] ; do echo 'Hello Marathon
' ; sleep 5 ; done","args":null,"user":null,"env":{},"instances"
:1,"cpus":0.1,"mem":10,"disk":0,"executor":"","constraints":[],"
uris":[],"storeUrls":[],"ports":[0],"requirePorts":false,"backof
fSeconds":1,"backoffFactor":1.15,"maxLaunchDelaySeconds":3600,"c
ontainer":null,"healthChecks":[],"dependencies":[],"upgradeStrat
egy":{"minimumHealthCapacity":1,"maximumOverCapacity":1},"labels"
:{},"acceptedResourceRoles":null,"version":"2015-12-28T05:33:05.
805Z","tasksStaged":0,"tasksRunning":0,"tasksHealthy":0,"tasksUn
healthy":0,"deployments":[{"id":"3ec3fbd5-11e4-479f-bd17-813d33e
43e0c"}],"tasks":[]}%
```

Marathon 的更多 REST API 可以参考本地自带的文档： `http://marathon_host:8080/api-console/index.html` 。

此时，如果运行任务的从节点出现故障，任务会自动在其它可用的从节点上启动。

此外，目前也已经支持基于 Docker 容器的任务。需要先在 Mesos slave 节点上为 slave 服务配置 `--containerizers=docker,mesos` 参数。

例如如下面的示例任务：

```
{
  "id": "basic-3",
  "cmd": "python3 -m http.server 8080",
  "cpus": 0.5,
  "mem": 32.0,
  "container": {
      "type": "DOCKER",
      "volumes": [],
      "docker": {
        "image": "python:3",
        "network": "BRIDGE",
        "portMappings": [
          {
            "containerPort": 8080,
            "hostPort": 31000,
            "servicePort": 0,
            "protocol": "tcp"
          }
        ],
        "privileged": false,
        "parameters": [],
        "forcePullImage": true
      }
  }
}
```

该任务启动一个 `python:3` 容器，执行 `python3 -m http.server 8080` 命令，作为一个简单的 web 服务，实际端口会映射到宿主机的 31000 端口。

注意区分 hostPort 和 servicePort，前者代表任务映射到的本地可用端口（可用范围由 Mesos slave 汇报，默认为 31000 ~ 32000）；后者作为服务管理的端口，可以被用作一些服务发行机制使用进行转发，在整个 Marathon 集群中是唯一的。

任务执行后，也可以在对应 slave 节点上通过 Docker 命令查看容器运行情况，容器将以 `mesos-SLAVE_ID` 开头。

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
   NAMES
1226b4ec8d7d        python:3            "/bin/sh -c 'python3 "
 3 days ago          Up 3 days           0.0.0.0:10000->8080/tcp
   mesos-06db0fba-49dc-4d28-ad87-6c2d5a020866-S10.b581149e-2c43-
46a2-b652-1a0bc10204b3
```

原理与架构

首先，再次需要强调 Mesos 自身只是一个资源调度框架，并非一整套完整的应用管理平台，所以只有 Mesos 自己是不能干活的。但是基于 Mesos，可以比较容易地为各种应用管理框架或者中间件平台（作为 Mesos 的应用）提供分布式运行能力；同时多个框架也可以同时运行在一个 Mesos 集群中，提高整体的资源使用效率。

Mesos 对自己定位范围的划分，使得它要完成的任务很明确，其它任务框架也可以很容易的与它进行整合。

架构

下面这张基本架构图来自 Mesos 官方。

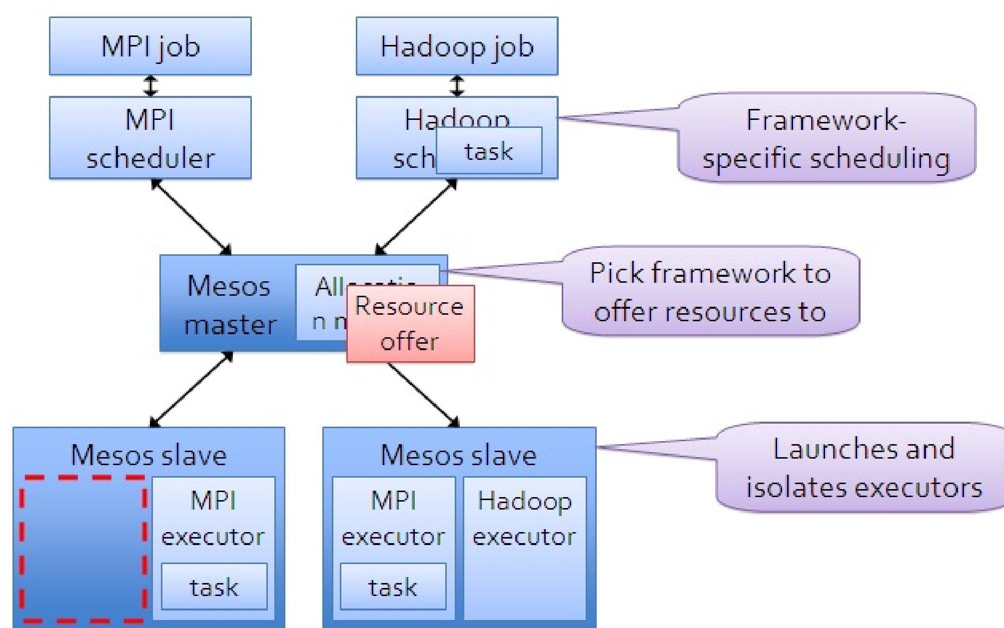


图 *1.21.3.1 - mesos* 的基本架构

可以看出，Mesos 采用了经典的主-从（master-slave）架构，其中主节点（管理节点）可以使用 zookeeper 来做 HA。

Mesos master 服务将运行在主节点上，Mesos slave 服务则需要运行在各个计算任务节点上。

负责完成具体任务的应用框架们，跟 Mesos master 进行交互，来申请资源。

基本单元

Mesos 中有三个基本的组件：管理服务（master）、任务服务（slave）以及应用框架（framework）。

管理服务 - master

跟大部分分布式系统中类似，主节点起到管理作用，将看到全局的信息，负责不同应用框架之间的资源调度和逻辑控制。应用框架需要注册到管理服务上才能被使用。

用户和应用需要通过主节点提供的 API 来获取集群状态和操作集群资源。

任务服务 - slave

负责汇报本从节点上的资源状态（空闲资源、运行状态等等）给主节点，并负责隔离本地资源来执行主节点分配的具体任务。

隔离机制目前包括各种容器机制，包括 LXC、Docker 等。

应用框架 - framework

应用框架是实际干活的，包括两个主要组件：

- 调度器（scheduler）：注册到主节点，等待分配资源；
- 执行器（executor）：在从节点上执行框架指定的任务（框架也可以使用 Mesos 自带的执行器，包括 shell 脚本执行器和 Docker 执行器）。

应用框架可以分两种：一种是对资源的需求是会扩展的（比如 Hadoop、Spark 等），申请后还可能调整；一种是对资源需求大小是固定的（MPI 等），一次申请即可。

调度

对于一个资源调度框架来说，最核心的就是调度机制，怎么能快速高效地完成对某个应用框架资源的分配，是核心竞争力所在。最理想情况下（大部分时候都无法实现），最好是能猜到应用们的实际需求，实现最大化的资源使用率。

Mesos 为了实现尽量优化的调度，采取了两层（two-layer）的调度算法。

算法基本过程

调度的基本思路很简单，master 先全局调度一大块资源给某个 framework，framework 自己再实现内部的细粒度调度，决定哪个任务用多少资源。两层调度简化了 Mesos master 自身的调度过程，通过将复杂的细粒度调度交由 framework 实现，避免了 Mesos master 成为性能瓶颈。

调度机制支持插件机制来实现不同的策略。默认是 Dominant Resource Fairness（DRF）。

注：*DRF* 算法细节可以参考论文《*Dominant Resource Fairness: Fair Allocation of Multiple Resource Types*》。其核心思想是对不同类型资源的多个请求，计算请求的主资源类型，然后根据主资源进行公平分配。

调度过程

调度通过 offer 发送的方式进行交互。一个 offer 是一组资源，例如 `<1 CPU, 2 GB Mem>` 。

基本调度过程如下：

- 首先，slave 节点会周期性汇报自己可用的资源给 master；
- 某个时候，master 收到应用框架发来的资源请求，根据调度策略，计算出来一个资源 offer 给 framework；
- framework 收到 offer 后可以决定要不要，如果接受的话，返回一个描述，说明自己希望如何使用和分配这些资源来运行某些任务（可以说明只希望使用部分资源，则多出来的会被 master 收回）；
- 最后，master 则根据 framework 答复的具体分配情况发送给 slave，以使用 framework 的 executor 来按照分配的资源策略执行任务。

具体给出一个例子，某从节点向主节点汇报自己有 `<4 CPU, 8 GB Mem>` 的空闲资源，同时，主节点看到某个应用框架请求 `<3 CPU, 6 GB Mem>` ，就创建一个 offer `<slave#1, 4 CPU, 8 GB Mem>` 把满足的资源发给应用框架。应用框架（的调度器）收到 offer 后觉得可以接受，就回复主节点，并告诉主节点希望运行两个任务：一个占用 `<1 CPU, 2 GB Mem>` ，一个占用 一个占用 `<2 CPU, 4 GB Mem>` 。主节点收到任务信息后分配任务到从节点上进行运行（实际上是应用框架的执行器来负责执行任务）。任务运行结束后资源可以被释放出来。

剩余的资源还可以继续分配给其他应用框架或任务。

应用框架在收到 offer 后，如果 offer 不满足自己的偏好（例如希望继续使用上次的 slave 节点），则可以选择拒绝 offer，等待 master 发送新的 offer 过来。另外，可以通过过滤器机制来加快资源的分配过程。

过滤器

framework 可以通过过滤器机制告诉 master 它的资源偏好，比如希望分配过来的 offer 有哪个资源，或者至少有多少资源等。

过滤器可以避免某些应用资源长期分配不到所需要的资源的情况，加速整个资源分配的交互过程。

回收机制

为了避免某些任务长期占用集群中资源，Mesos 也支持回收机制。

主节点可以定期回收计算节点上的任务所占用的资源，可以动态调整长期任务和短期任务的分布。

HA

从架构上看，最为核心的节点是 master 节点。除了使用 ZooKeeper 来解决单点失效问题之外，Mesos 的 master 节点自身还提供了很高的鲁棒性。

Mesos master 节点在重启后，可以动态通过 slave 和 framework 发来的消息重建内部状态，虽然可能导致一定的时延，但这避免了传统控制节点对数据库的依赖。

当然，为了减少 master 节点的负载过大，在集群中 slave 节点数目较多的时候，要避免把各种通知的周期配置的过短。实践中，可以通过部署多个 Mesos 集群来保持单个集群的规模不要过大。

Mesos 配置项解析

Mesos 支持在运行时通过命令行参数形式提供的配置项。如果是通过系统服务方式启动，也支持以配置文件或环境变量方式给出。当然，实际上最终是提取为命令行参数传递给启动命令。

Mesos 的配置项分为三种类型：通用项（master 和 slave 都支持），只有 master 支持的，以及只有 slave 支持的。

Mesos 配置项比较多，下面对一些重点配置进行描述。少数为必备项，意味着必须给出配置值；另外一些是可选配置，自己带有默认值。

通用项

通用项数量不多，主要涉及到服务绑定地址和日志信息等，包括：

- `--advertise_ip=VALUE` 可以通过该地址访问到服务，比如应用框架访问到 master 节点；
- `--advertise_port=VALUE` 可以通过该端口访问到服务；
- `--external_log_file=VALUE` 指定存储日志的外部文件，可通过 Web 界面查看；
- `--firewall_rules=VALUE` endpoint 防火墙规则， `VALUE` 可以是 JSON 格式或者存有 JSON 格式的文件路径；
- `--ip=VALUE` 服务绑定到的IP 地址，用来监听外面过来的请求；
- `--log_dir=VALUE` 日志文件路径，如果为空（默认值）则不存储日志到本地；
- `--logbufsecs=VALUE` buffer 多少秒的日志，然后写入本地；
- `--logging_level=VALUE` 日志记录的最低级别；
- `--port=VALUE` 绑定监听的端口，master 默认是 5050，slave 默认是 5051。

master 专属配置项

这些配置项是针对主节点上的 Mesos master 服务的，围绕高可用、注册信息、对应用框架的资源管理等。用户应该根据本地主节点资源情况来合理的配置这些选项。

用户可以通过 `mesos-master --help` 命令来获取所有支持的配置项信息。

必须指定的配置项有三个：

- `--quorum=VALUE` 必备项，使用基于 replicated-Log 的注册表（即利用 ZooKeeper 实现 HA）时，参与投票时的最少节点个数；
- `--work_dir=VALUE` 必备项，注册表持久化信息存储位置；
- `--zk=VALUE` 如果主节点为 HA 模式，此为必备项，指定 ZooKeepr 的服务地址，支持多个地址，之间用逗号隔离，例如 `zk://username:password@host1:port1,host2:port2,.../path` 。还可以为存有路径信息的文件路径。

可选的配置项有：

- `--acls=VALUE` ACL 规则或所在文件；
- `--allocation_interval=VALUE` 执行 allocation 的间隔，默认为 1sec；
- `--allocator=VALUE` 分配机制，默认为 HierarchicalDRF；
- `--[no-]authenticate` 是否允许非认证过的 framework 注册；
- `--[no-]authenticate_slaves` 是否允许非认证过的 slaves 注册；
- `--authenticators=VALUE` 对 framework 或 salves 进行认证时的实现机制；
- `--cluster=VALUE` 集群别名，显示在 Web 界面上供用户识别的；
- `--credentials=VALUE` 存储加密后凭证的文件的路径；
- `--external_log_file=VALUE` 采用外部的日志文件；
- `--framework_sorter=VALUE` 给定 framework 之间的资源分配策略；
- `--hooks=VALUE` master 中安装的 hook 模块；
- `--hostname=VALUE` master 节点使用的主机名，不配置则从系统中获取；
- `--[no-]log_auto_initialize` 是否自动初始化注册表需要的 replicated 日志；
- `--modules=VALUE` 要加载的模块，支持文件路径或者 JSON；
- `--offer_timeout=VALUE` offer 撤销的超时；
- `--rate_limits=VALUE` framework 的速率限制，即 query per second (qps)；
- `--recovery_slave_removal_limit=VALUE` 限制注册表恢复后可以移除或停止的 slave 数目，超出后 master 会失败，默认是 100%；
- `--slave_removal_rate_limit=VALUE slave` 没有完成健康度检查时候被移除的速率上限，例如 1/10mins 代表每十分钟最多有一个；
- `--registry=VALUE` 注册表信息的持久化策略，默认为 `replicated_log`

存放本地，还可以为 `in_memory` 放在内存中；
- `--registry_fetch_timeout=VALUE` 访问注册表失败超时；
- `--registry_store_timeout=VALUE` 存储注册表失败超时；
- `--[no-]registry_strict` 是否按照注册表中持久化信息执行操作，默认为 false；
- `--roles=VALUE` 集群中 framework 可以所属的分配角色；
- `--[no-]root_submissions` root 是否可以提交 framework，默认为 true；
- `--slave_reregister_timeout=VALUE` 新的 lead master 节点选举出来后，多久之内所有的 slave 需要注册，超时的 salve 将被移除并关闭，默认为 10mins；
- `--user_sorter=VALUE` 在用户之间分配资源的策略，默认为 drf；
- `--webui_dir=VALUE` webui 实现的文件目录所在，默认为 `/usr/local/share/mesos/webui` ；
- `--weights=VALUE` 各个角色的权重；
- `--whitelist=VALUE` 文件路径，包括发送 offer 的 slave 名单，默认为 None；
- `--zk_session_timeout=VALUE` session 超时，默认为 10secs；
- `--max_executors_per_slave=VALUE` 配置了 `--with-network-isolator` 时可用，限制每个 slave 同时执行任务个数。

下面给出一个由三个节点组成的 master 集群典型配置，工作目录指定为 `/tmp/mesos` ，集群名称为 `mesos_cluster` 。

```
mesos-master \
--zk=zk://10.0.0.2:2181,10.0.0.3:2181,10.0.0.4:2181/mesos \
--quorum=2 \
--work_dir=/tmp/mesos \
--cluster=mesos_cluster
```

slave 专属配置项

slave 节点支持的配置项是最多的，因为它所完成的事情也最复杂。这些配置项既包括跟主节点打交道的一些参数，也包括对本地资源的配置，包括隔离机制、本地任务的资源限制等。

用户可以通过 `mesos-slave --help` 命令来获取所有支持的配置项信息。

必备项就一个：

- `--master=VALUE` 必备项，master 所在地址，或对应 ZooKeeper 服务地址，或文件路径，可以是列表。

以下为可选配置项：

- `--attributes=VALUE` 机器属性；
- `--authenticatee=VALUE` 跟 master 进行认证时候的认证机制；
- `--[no-]cgroups_enable_cfs` 采用 CFS 进行带宽限制时候对 CPU 资源进行限制，默认为 false；
- `--cgroups_hierarchy=VALUE` cgroups 的目录根位置，默认为 `/sys/fs/cgroup` ；
- `--[no-]cgroups_limit_swap` 限制内存和 swap，默认为 false，只限制内存；
- `--cgroups_root=VALUE` 根 cgroups 的名称，默认为 mesos；
- `--container_disk_watch_interval=VALUE` 为容器进行硬盘配额查询的时间间隔；
- `--containerizer_path=VALUE` 采用外部隔离机制（ `--isolation=external` ）时候，外部容器机制执行文件路径；
- `--containerizers=VALUE` 可用的容器实现机制，包括 mesos、external、docker；
- `--credential=VALUE` 加密后凭证，或者所在文件路径；
- `--default_container_image=VALUE` 采用外部容器机制时，任务缺省使用的镜像；
- `--default_container_info=VALUE` 容器信息的缺省值；
- `--default_role=VALUE` 资源缺省分配的角色；
- `--disk_watch_interval=VALUE` 硬盘使用情况的周期性检查间隔，默认为 1mins；
- `--docker=VALUE` docker 执行文件的路径；
- `--docker_remove_delay=VALUE` 删除容器之前的等待时间，默认为 6hrs；
- `--[no-]docker_kill_orphans` 清除孤儿容器，默认为 true；
- `--docker_sock=VALUE` docker sock 地址，默认为 `/var/run/docker.sock` ；
- `--docker_mesos_image=VALUE` 运行 slave 的 docker 镜像，如果被配置，docker 会假定 slave 运行在一个 docker 容器里；
- `--docker_sandbox_directory=VALUE` sandbox 映射到容器里的哪个路

径；

- `--docker_stop_timeout=VALUE` 停止实例后等待多久执行 kill 操作，默认为 0secs；
- `--[no-]enforce_container_disk_quota` 是否启用容器配额限制，默认为 false；
- `--executor_registration_timeout=VALUE` 执行应用最多可以等多久再注册到 slave，否则停止它，默认为 1mins；
- `--executor_shutdown_grace_period=VALUE` 执行应用停止后，等待多久，默认为 5secs；
- `--external_log_file=VALUE` 外部日志文件；
- `--fetcher_cache_size=VALUE` fetcher 的 cache 大小，默认为 2 GB；
- `--fetcher_cache_dir=VALUE` fetcher cache 文件存放目录，默认为 /tmp/mesos/fetch；
- `--frameworks_home=VALUE` 执行应用前添加的相对路径，默认为空；
- `--gc_delay=VALUE` 多久清理一次执行应用目录，默认为 1weeks；
- `--gc_disk_headroom=VALUE` 调整计算最大执行应用目录年龄的硬盘留空量，默认为 0.1；
- `--hadoop_home=VALUE` hadoop 安装目录，默认为空，会自动查找 HADOOP_HOME 或者从系统路径中查找；
- `--hooks=VALUE` 安装在 master 中的 hook 模块列表；
- `--hostname=VALUE` slave 节点使用的主机名；
- `--isolation=VALUE` 隔离机制，例如 `posix/cpu,posix/mem` （默认）或者 `cgroups/cpu,cgroups/mem` 、 `external` 等；
- `--launcher_dir=VALUE` mesos 可执行文件的路径，默认为 `/usr/local/lib/mesos` ；
- `--image_providers=VALUE` 支持的容器镜像机制，例如 'APPC,DOCKER'；
- `--oversubscribed_resources_interval=VALUE` slave 节点定期汇报超配资源状态的周期；
- `--modules=VALUE` 要加载的模块，支持文件路径或者 JSON；
- `--perf_duration=VALUE` perf 采样时长，必须小于 perf_interval，默认为 10secs；
- `--perf_events=VALUE` perf 采样的事件；
- `--perf_interval=VALUE` perf 采样的时间间隔；
- `--qos_controller=VALUE` 超配机制中保障 QoS 的控制器名；
- `--qos_correction_interval_min=VALUE` Qos 控制器纠正超配资源的最小

间隔，默认为 0secs；

- `--recover=VALUE` 回复后是否重连旧的执行应用，reconnect（默认值）是重连，cleanup 清除旧的执行器并退出；
- `--recovery_timeout=VALUE` slave 恢复时的超时，太久则所有相关的执行应用将自行退出，默认为 15mins；
- `--registration_backoff_factor=VALUE` 跟 master 进行注册时候的重试时间间隔算法的因子，默认为 1secs，采用随机指数算法，最长 1mins；
- `--resource_monitoring_interval=VALUE` 周期性监测执行应用资源使用情况的间隔，默认为 1secs；
- `--resources=VALUE` 每个 slave 可用的资源，比如主机端口默认为 [31000, 32000]；
- `--[no-]revocable_cpu_low_priority` 运行在可撤销 CPU 上容器将拥有较低优先级，默认为 true。
- `--slave_subsystems=VALUE` slave 运行在哪些 cgroup 子系统中，包括 memory，cpuacct 等，缺省为空；
- `--[no-]strict` 是否认为所有错误都不可忽略，默认为 true；
- `--[no-]switch_user` 用提交任务的用户身份来运行，默认为 true；
- `--work_dir=VALUE` framework 的工作目录，默认为 /tmp/mesos。

下面这些选项需要配置 `--with-network-isolator` 一起使用（编译时需要启用 `--with-network-isolator` 参数）。

- `--ephemeral_ports_per_container=VALUE` 分配给一个容器的临时端口的最大数目，需要为 2 的整数幂（默认为 1024）；
- `--eth0_name=VALUE` public 网络的接口名称，如果不指定，根据主机路由进行猜测；
- `--lo_name=VALUE` loopback 网卡名称；
- `--egress_rate_limit_per_container=VALUE` 每个容器的输出流量限制速率限制（采用 fq_codel 算法来限速），单位是字节每秒；
- `--[no-]-egress_unique_flow_per_container` 是否把不同容器的流量当作彼此不同的流，避免彼此影响（默认为 false）；
- `--[no-]network_enable_socket_statistics` 是否采集每个容器的 socket 统计信息，默认为 false。

下面给出一个典型的 slave 配置，容器为 Docker，监听在 `10.0.0.10` 地址；节点上限制 16 个 CPU、64 GB 内存，容器的非临时端口范围指定为 [31000-32000]，临时端口范围指定为 [32768-57344]；每个容器临时端口最多为 512 个，

并且外出流量限速为 50 MB/s。

```
mesos-slave \
--master=zk://10.0.0.2:2181,10.0.0.3:2181,10.0.0.4:2181/mesos \
--containerizers=docker \
--ip=10.0.0.10 \
--isolation=cgroups/cpu,cgroups/mem,network/port_mapping \
--resources=cpus:16;mem:64000;ports:[31000-32000];ephemeral_port
s:[32768-57344] \
--ephemeral_ports_per_container=512 \
--egress_rate_limit_per_container=50000KB \
--egress_unique_flow_per_container
```

为了避免主机分配的临时端口跟我们指定的临时端口范围冲突，需要在主机节点上进行配置。

```
$ echo "57345 61000" > /proc/sys/net/ipv4/ip_local_port_range
```

注：非临时端口是 *Mesos* 分配给框架，绑定到任务使用的，端口号往往有明确意义；临时端口是系统分配的，往往不太关心具体端口号。

日志与监控

Mesos 自身提供了强大的日志和监控功能，某些应用框架也提供了针对框架中任务的监控能力。通过这些接口，用户可以实时获知集群的各种状态。

日志配置

日志文件默认在 `/var/log/mesos` 目录下，根据日志等级带有不同后缀。

用户可以通过日志来调试使用中碰到的问题。

一般的，推荐使用 `--log_dir` 选项来指定日志存放路径，并通过日志分析引擎来进行监控。

监控

Mesos 提供了方便的监控接口，供用户查看集群中各个节点的状态。

主节点

通过 `http://MASTER_NODE:5050/metrics/snapshot` 地址可以获取到 Mesos 主节点的各种状态统计信息，包括资源（CPU、硬盘、内存）使用、系统状态、从节点、应用框架、任务状态等。

例如查看主节点 `10.0.0.2` 的状态信息，并用 jq 来解析返回的 json 对象。

```
$ curl -s http://10.0.0.2:5050/metrics/snapshot |jq .
{
  "system/mem_total_bytes": 4144713728,
  "system/mem_free_bytes": 153071616,
  "system/load_5min": 0.37,
  "system/load_1min": 0.6,
  "system/load_15min": 0.29,
  "system/cpus_total": 4,
  "registrar/state_store_ms/p9999": 45.4096616192,
  "registrar/state_store_ms/p999": 45.399272192,
  "registrar/state_store_ms/p99": 45.29537792,
  "registrar/state_store_ms/p95": 44.8336256,
  "registrar/state_store_ms/p90": 44.2564352,
  "registrar/state_store_ms/p50": 34.362368,
  ...
  "master/recovery_slave_removals": 1,
  "master/slave_registrations": 0,
  "master/slave_removals": 0,
  "master/slave_removals/reason_registered": 0,
  "master/slave_removals/reason_unhealthy": 0,
  "master/slave_removals/reason_unregistered": 0,
  "master/slave_reregistrations": 2,
  "master/slave_shutdowns_canceled": 0,
  "master/slave_shutdowns_completed": 1,
  "master/slave_shutdowns_scheduled": 1
}
```

从节点

通过 `http://SLAVE_NODE:5051/metrics/snapshot` 地址可以获取到 Mesos 从节点的各种状态统计信息，包括资源、系统状态、各种消息状态等。

例如查看从节点 `10.0.0.10` 的状态信息。

```
$ curl -s http://10.0.0.10:5051/metrics/snapshot |jq .
{
  "system/mem_total_bytes": 16827785216,
  "system/mem_free_bytes": 3377315840,
  "system/load_5min": 0.11,
```

```
    "system/load_1min": 0.16,
    "system/load_15min": 0.13,
    "system/cpus_total": 8,
    "slave/valid_status_updates": 11,
    "slave/valid_framework_messages": 0,
    "slave/uptime_secs": 954125.458927872,
    "slave/tasks_starting": 0,
    "slave/tasks_staging": 0,
    "slave/tasks_running": 1,
    "slave/tasks_lost": 0,
    "slave/tasks_killed": 2,
    "slave/tasks_finished": 0,
    "slave/executors_preempted": 0,
    "slave/executor_directory_max_allowed_age_secs": 403050.709525
191,
    "slave/disk_used": 0,
    "slave/disk_total": 88929,
    "slave/disk_revocable_used": 0,
    "slave/disk_revocable_total": 0,
    "slave/disk_revocable_percent": 0,
    "slave/disk_percent": 0,
    "containerizer/mesos/container_destroy_errors": 0,
    "slave/container_launch_errors": 6,
    "slave/cpus_percent": 0.025,
    "slave/cpus_revocable_percent": 0,
    "slave/cpus_revocable_total": 0,
    "slave/cpus_revocable_used": 0,
    "slave/cpus_total": 8,
    "slave/cpus_used": 0.2,
    "slave/executors_registering": 0,
    "slave/executors_running": 1,
    "slave/executors_terminated": 8,
    "slave/executors_terminating": 0,
    "slave/frameworks_active": 1,
    "slave/invalid_framework_messages": 0,
    "slave/invalid_status_updates": 0,
    "slave/mem_percent": 0.00279552715654952,
    "slave/mem_revocable_percent": 0,
    "slave/mem_revocable_total": 0,
    "slave/mem_revocable_used": 0,
```

```
  "slave/mem_total": 15024,
  "slave/mem_used": 42,
  "slave/recovery_errors": 0,
  "slave/registered": 1,
  "slave/tasks_failed": 6
}
```

另外，通过 `http://MASTER_NODE:5050/monitor/statistics.json` 地址可以看到该从节点上容器网络相关的统计数据，包括进出流量、丢包数、队列情况等。获取方法同上，在此不再演示。

常见应用框架

应用框架是实际干活的，可以理解为 Mesos 之上跑的 `应用` 。应用框架注册到 Mesos master 服务上即可使用。

用户大部分时候，只需要跟应用框架打交道。因此，选择合适的应用框架十分关键。

Mesos 目前支持的应用框架分为四大类：长期运行任务（以及 PaaS）、大数据处理、批量调度、数据存储。

随着 Mesos 自身的发展，越来越多的框架开始支持 Mesos，下面总结了目前常用的一些框架。

长期运行的服务

Aurora

项目维护地址在 http://aurora.incubator.apache.org。

利用 mesos 调度安排的任务，保证任务一直在运行。

提供 REST 接口，客户端和 webUI（8081 端口）

Marathon

项目维护地址在 https://github.com/mesosphere/marathon。

一个私有 PaaS 平台，保证运行的应用不被中断。

如果任务停止了，会自动重启一个新的相同任务。

支持任务为任意 bash 命令，以及容器。

提供 REST 接口，客户端和 webUI（8080 端口）

Singularity

项目维护地址在 https://github.com/HubSpot/Singularity。

一个私有 PaaS 平台。

调度器，运行长期的任务和一次性任务。

提供 REST 接口，客户端和 webUI（7099、8080 端口），支持容器。

大数据处理

Cray Chapel

项目维护地址在 https://github.com/nqn/mesos-chapel。

支持 Chapel 并行编程语言的运行框架。

Dpark

项目维护地址在 https://github.com/douban/dpark。

Spark 的 Python 实现。

Hadoop

项目维护地址在 https://github.com/mesos/hadoop。

经典的 map-reduce 模型的实现。

Spark

项目维护地址在 http://spark.incubator.apache.org。

跟 Hadoop 类似，但处理迭代类型任务会更好的使用内存做中间状态缓存，速度要快一些。

Storm

项目维护地址在 https://github.com/mesosphere/storm-mesos。

分布式流计算，可以实时处理数据流。

批量调度

Chronos

项目维护地址在 https://github.com/airbnb/chronos。

Cron 的分布式实现，负责任务调度，支持容错。

Jenkins

项目维护地址在 https://github.com/jenkinsci/mesos-plugin。

大名鼎鼎的 CI 引擎。使用 mesos-jenkins 插件，可以将 jenkins 的任务被 Mesos 集群来动态调度执行。

JobServer

项目维护地址在 http://www.grandlogic.com/content/html_docs/jobserver.html。

基于 Java 的调度任务和数据处理引擎。

GoDocker

项目维护地址在 https://bitbucket.org/osallou/go-docker。

基于 Docker 容器的集群维护工具。提供用户接口，除了支持 Mesos，还支持 Kubernetes、Swarm 等。

数据存储

Cassandra

项目维护地址在 https://github.com/mesosphere/cassandra-mesos。

高性能的分布式数据库。可扩展性很好，支持高可用。

ElasticSearch

项目维护地址在 https://github.com/mesosphere/elasticsearch-mesos。

功能十分强大的分布式数据搜索引擎。

一方面通过分布式集群实现可靠的数据库，一方面提供灵活的 API，对数据进行整合和分析。ElasticSearch + LogStash + Kibana 目前合成为 ELK 工具栈。

Hypertable

项目维护地址在 https://code.google.com/p/hypertable。

高性能的分布式数据库，支持结构化或者非结构化的数据存储。

Tachyon

项目维护地址在 http://tachyon-project.org/。

内存为中心的分布式存储系统，利用内存访问的高速提供高性能。

本章小结

本章讲解了 Mesos 的安装使用、基本原理和架构，以及支持 Mesos 的重要应用框架。Mesos 最初设计为资源调度器，然而其灵活的设计和对上层框架的优秀支持，使得它可以很好的支持大规模的分布式应用场景。结合 Docker，Mesos 可以很容易部署一套私有的容器云。

除了核心功能之外，Mesos 在设计上有许多值得借鉴之处，比如它清晰的定位、简洁的架构、细致的参数、高度容错的可靠，还有对限速、监控等的支持等。

Mesos 作为一套成熟的开源项目，可以很好的被应用和集成到生产环境中。但它的定位集中在资源调度，往往需要结合应用框架或二次开发。

容器与云计算

Docker 目前已经得到了众多公有云平台的支持，并成为除虚拟机之外的核心云业务。

除了 AWS、Google、Azure、Docker 官方云服务等，国内的各大公有云厂商，基本上都同时支持了虚拟机服务和容器服务，甚至还专门推出了容器云业务。

简介

目前与容器相关的云计算主要分为两种类型。

一种是传统的 IaaS 服务商提供对容器相关的服务，包括镜像下载、容器托管等。

另一种是直接基于容器技术对外提供容器云服务，所谓 Container as a Service（CaaS）。

亚马逊云



图 *1.22.2.1 - AWS*

AWS，即 Amazon Web Services，是亚马逊（Amazon）公司的 IaaS 和 PaaS 平台服务。AWS 提供了一整套基础设施和应用程序服务，使用户几乎能够在云中运行一切应用程序：从企业应用程序和大数据项目，到社交游戏和移动应用程序。AWS 面向用户提供包括弹性计算、存储、数据库、应用程序在内的一整套云计算服务，能够帮助企业降低 IT 投入成本和维护成本。

自 2006 年初起，亚马逊 AWS 开始在云中为各种规模的公司提供技术服务平台。利用亚马逊 AWS，软件开发人员可以轻松购买计算、存储、数据库和其他基于 Internet 的服务来支持其应用程序。开发人员能够灵活选择任何开发平台或编程环境，以便于其尝试解决问题。由于开发人员只需按使用量付费，无需前期资本支出，亚马逊 AWS 是向最终用户交付计算资源、保存的数据和其他应用程序的一种经济划算的方式。

2015 年 AWS 正式发布了 EC2 容器服务(ECS)。ECS 的目的是让 Docker 容器变的更加简单，它提供了一个集群和编排的层，用来控制主机上的容器部署，以及部署之后的集群内的容器的生命周期管理。ECS 是诸如 Docker Swarm、Kubernetes、Mesos 等工具的替代，它们工作在同一个层，除了作为一个服务来提供。这些工具和 ECS 不同的地方在于，前者需要用户自己来部署和管理，而 ECS 是“作为服务”来提供的。

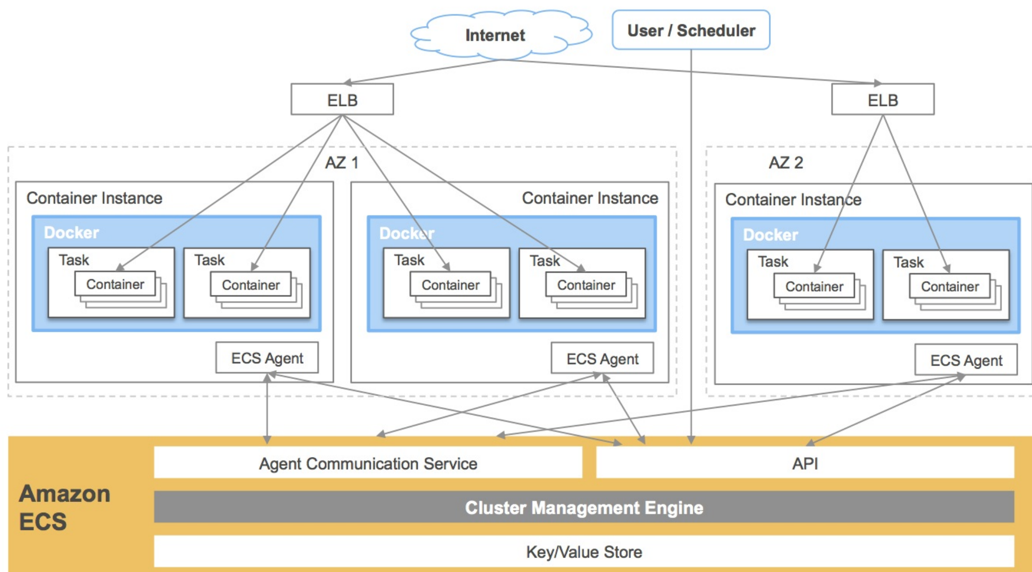


图 *1.22.2.2 - AWS* 容器服务

腾讯云



图 *1.22.3.1* - 腾讯云

腾讯云在架构方面经过多年积累，并且有着多年对海量互联网服务的经验。不管是社交、游戏还是其他领域，都有多年的成熟产品来提供产品服务。腾讯在云端完成重要部署，为开发者及企业提供云服务、云数据、云运营等整体一站式服务方案。

具体包括云服务器、云存储、云数据库和弹性 web 引擎等基础云服务；腾讯云分析（MTA）、腾讯云推送（信鸽）等腾讯整体大数据能力；以及 QQ互联、QQ 空间、微云、微社区等云端链接社交体系。这些正是腾讯云可以提供给这个行业的差异化优势，造就了可支持各种互联网使用场景的高品质的腾讯云技术平台。

腾讯云容器服务是高度可扩展的高性能容器管理服务，用户可以在托管的云服务器实例集群上轻松运行应用程序。使用该服务，将无需安装、运维、扩展用户的集群管理基础设施，只需进行简单的API调用，便可启动和停止Docker应用程序，查询集群的完整状态，以及使用各种云服务。用户可以根据用户的资源需求和可用性要求在用户的集群中安排容器的置放，满足业务或应用程序的特定要求。

腾讯云已有用户或新用户都可以 通过这里，在不做任何迁移的前提下，获得腾讯云服务的额外支持和大客户专享价格。

阿里云



图 *1.22.4.1* - 阿里云

阿里云创立于 2009 年，是中国较早的云计算平台。阿里云致力于提供安全、可靠的计算和数据处理能力。

阿里云的客户群体中，活跃着微博、知乎、魅族、锤子科技、小咖秀等一大批明星互联网公司。在天猫双 11 全球狂欢节等极富挑战的应用场景中，阿里云保持着良好的运行纪录。

阿里云容器服务提供了高性能、可伸缩的容器应用管理服务，支持在一组云服务器上通过 Docker 容器来进行应用生命周期管理。容器服务极大简化了用户对容器管理集群的搭建工作，无缝整合了阿里云虚拟化、存储、网络和安全能力。容器服务提供了多种应用发布方式和流水线般的持续交付能力，原生支持微服务架构，助力用户无缝上云和跨云管理。

本章小结

本章介绍了公有云服务对 Docker 的积极支持，以及新出现的容器云平台。

事实上，Docker 技术的出现自身就极大推动了云计算行业的发展。

通过整合公有云的虚拟机和 Docker 方式，可能获得更多的好处，包括

- 更快速的持续交付和部署能力；
- 利用内核级虚拟化，对公有云中服务器资源进行更加高效地利用；
- 利用公有云和 Docker 的特性更加方便的迁移和扩展应用。

同时，容器将作为与虚拟机类似的业务直接提供给用户使用，极大的丰富了应用开发和部署的场景。

操作系统

目前常用的 Linux 发行版主要包括 Debian/Ubuntu 系列和 CentOS/Fedora 系列。

前者以自带软件包版本较新而出名；后者则宣称运行更稳定一些。选择哪个操作系统取决于读者的具体需求。同时，社区还推出了完全基于 Docker 的 Linux 发行版 CoreOS。

使用 Docker，读者只需要一个命令就能快速获取一个 linux 发行版镜像，这是以往包括各种虚拟化技术都难以实现的。这些镜像一般都很精简，但是可以支持完整 linux 系统的大部分功能。

本章将介绍如何使用 Docker 安装和使用 Busybox、Alphine、Debian/Ubuntu、CentOS/Fedora、CoreOS 等操作系统。

Busybox

简介

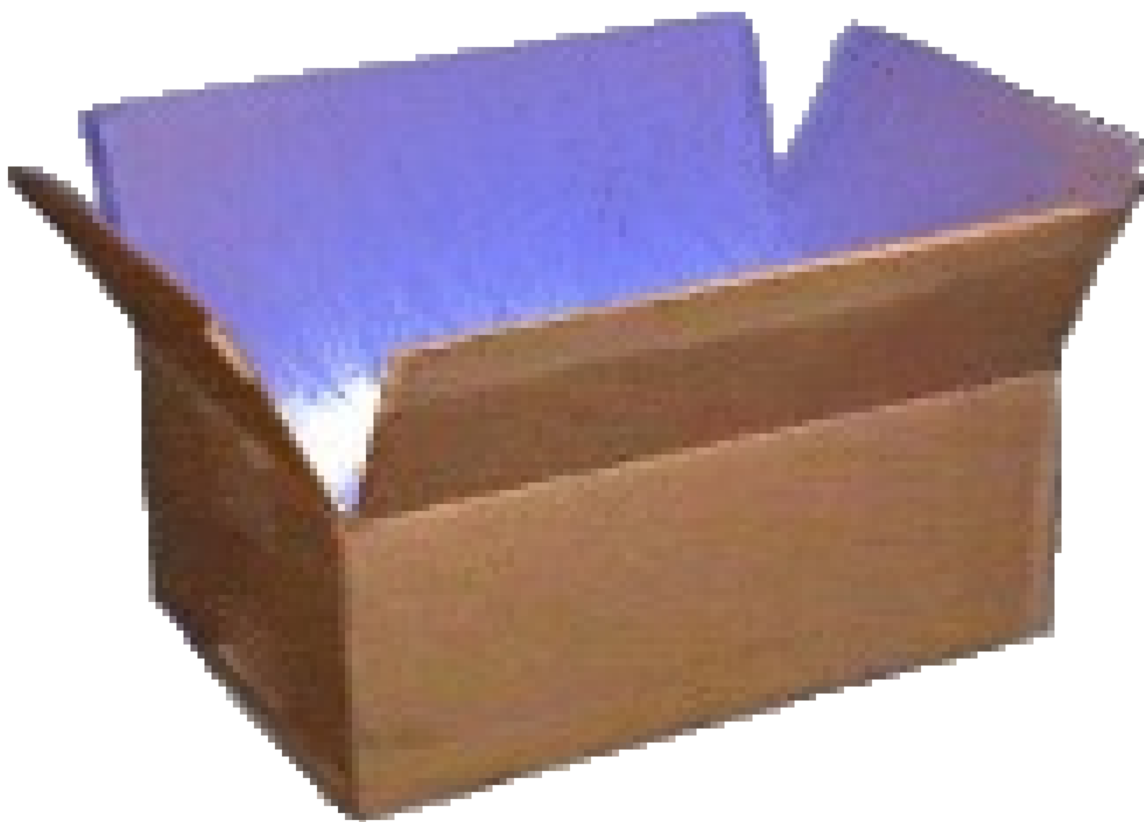


图 *1.23.1.1 - Busybox - Linux* 瑞士军刀

BusyBox 是一个集成了一百多个最常用 Linux 命令和工具（如 cat、echo、grep、mount、telnet 等）的精简工具箱，它只需要几 MB 的大小，很方便进行各种快速验证，被誉为“Linux 系统的瑞士军刀”。

BusyBox 可运行于多款 POSIX 环境的操作系统中，如 Linux（包括 Android）、Hurd、FreeBSD 等。

获取官方镜像

在 Docker Hub 中搜索 busybox 相关的镜像。

```
$ docker search busybox
NAME                                DESCRIPTION
                STARS     OFFICIAL   AUTOMATED
busybox                             Busybox base image.
                755       [OK]
progrium/busybox
                63                   [OK]
radial/busyboxplus                  Full-chain, Internet enabled, bu
sybox made...   11                   [OK]
odise/busybox-python
                3                    [OK]
multiarch/busybox                   multiarch ports of ubuntu-deboot
strap           2                    [OK]
azukiapp/busybox                    This image is meant to be used a
s the base...   2                    [OK]
...
```

读者可以看到最受欢迎的镜像同时带有 OFFICIAL 标记，说明它是官方镜像。用户使用 docker pull 指令下载镜像 `busybox:latest` ：

```
$ docker pull busybox:latest
busybox:latest: The image you are pulling has been verified
e433a6c5b276: Pull complete
e72ac664f4f0: Pull complete
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
Status: Downloaded newer image for busybox:latest
```

下载后，可以看到 busybox 镜像只有2.433 MB：

```
$ docker images
REPOSITORY                     TAG                   IMAGE ID
     CREATED                VIRTUAL SIZE
busybox                         latest            e72ac664f4f0
  6 weeks ago         2.433 MB
```

运行 busybox

启动一个 busybox 容器，并在容器中执行 grep 命令。

```
$ docker run -it busybox
/ # grep
BusyBox v1.22.1 (2014-05-22 23:22:11 UTC) multi-call binary.

Usage: grep [-HhnlLoqvsriwFE] [-m N] [-A/B/C N] PATTERN/-e PATTE
RN.../-f FILE [FILE]...

Search for PATTERN in FILEs (or stdin)

        -H      Add 'filename:' prefix
        -h      Do not add 'filename:' prefix
        -n      Add 'line_no:' prefix
        -l      Show only names of files that match
        -L      Show only names of files that don't match
        -c      Show only count of matching lines
        -o      Show only the matching part of line
        -q      Quiet. Return 0 if PATTERN is found, 1 otherwise
        -v      Select non-matching lines
        -s      Suppress open and read errors
        -r      Recurse
        -i      Ignore case
        -w      Match whole words only
        -x      Match whole lines only
        -F      PATTERN is a literal (not regexp)
        -E      PATTERN is an extended regexp
        -m N    Match up to N times per file
        -A N    Print N lines of trailing context
        -B N    Print N lines of leading context
        -C N    Same as '-A N -B N'
        -e PTRN Pattern to match
        -f FILE Read pattern from file
```

查看容器内的挂载信息。

```
/ # mount
rootfs on / type rootfs (rw)
none on / type aufs (rw,relatime,si=b455817946f8505c)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,mode=755)
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size
=65536k)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,
mode=620,ptmxmode=666)
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
/dev/disk/by-uuid/b1f2dba7-d91b-4165-a377-bf1a8bed3f61 on /etc/r
esolv.conf type ext4 (rw,relatime,errors=remount-ro,data=ordered
)
/dev/disk/by-uuid/b1f2dba7-d91b-4165-a377-bf1a8bed3f61 on /etc/h
ostname type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/dev/disk/by-uuid/b1f2dba7-d91b-4165-a377-bf1a8bed3f61 on /etc/h
osts type ext4 (rw,relatime,errors=remount-ro,data=ordered)
devpts on /dev/console type devpts (rw,nosuid,noexec,relatime,gi
d=5,mode=620,ptmxmode=000)
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,re
latime)
proc on /proc/irq type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,mode=755)
```

busybox 镜像虽然小巧，但包括了大量常见的 Linux 命令，读者可以用它快速熟悉 Linux 命令。

相关资源

- `Busybox` 官网：`https://busybox.net/`
- `Busybox` 官方仓库：`https://git.busybox.net/busybox/`
- `Busybox` 官方镜像：`https://hub.docker.com/_/busybox/`
- `Busybox` 官方仓库：`https://github.com/docker-library/busybox`

Alpine

简介



图 *1.23.2.1 - Apline Linux* 操作系统

`Alpine` 操作系统是一个面向安全的轻型 `Linux` 发行版。它不同于通常 `Linux` 发行版，`Alpine` 采用了 `musl libc` 和 `busybox` 以减小系统的体积和运行时资源消耗，但功能上比 `busybox` 又完善的多，因此得到开源社区越来越多的青睐。在保持瘦身的同时，`Alpine` 还提供了自己的包管理工具 `apk`，可以通过 `https://pkgs.alpinelinux.org/packages` 网站上查询包信息，也可以直接通过 `apk` 命令直接查询和安装各种软件。

`Alpine` 由非商业组织维护的，支持广泛场景的 `Linux` 发行版，它特别为资深/重度 `Linux` 用户而优化，关注安全，性能和资源效能。`Alpine` 镜像可以适用于更多常用场景，并且是一个优秀的可以适用于生产的基础系统/环境。

`Alpine` Docker 镜像也继承了 Alpine Linux 发行版的这些优势。相比于其他 `Docker` 镜像，它的容量非常小，仅仅只有 5 MB 左右（对比 Ubuntu 系列镜像接近 200 MB），且拥有非常友好的包管理机制。官方镜像来自 `docker-alpine` 项目。

目前 Docker 官方已开始推荐使用 `Alpine` 替代之前的 `Ubuntu` 做为基础镜像环境。这样会带来多个好处。包括镜像下载速度加快，镜像安全性提高，主机之间的切换更方便，占用更少磁盘空间等。

下表是官方镜像的大小比较：

```
REPOSITORY          TAG           IMAGE ID          VIRTUAL SIZE
alpine              latest        4e38e38c8ce0      4.799 MB
debian              latest        4d6ce913b130      84.98 MB
ubuntu              latest        b39b81afc8ca      188.3 MB
centos              latest        8efe422e6104      210 MB
```

获取并使用官方镜像

由于镜像很小，下载时间往往很短，读者可以直接使用 `docker run` 指令直接运行一个 `Alpine` 容器，并指定运行的 Linux 指令，例如：

```
$ docker run alpine echo '123'
123
```

笔者使用 time 工具来测试下在本地没有提前 pull 镜像情况下，执行 echo 命令的时间，仅需要 3 秒左右。

```
$ time docker run alpine echo '123'Unable to find image 'alpine:
latest' locallylatest: Pulling from library/alpine

e110a4a17941: Pull completeDigest: sha256:3dcdb92d7432d56604d454
5cbd324b14e647b313626d99b889d0626de158f73aStatus: Downloaded new
er image for alpine:latest123

real 0m3.367s user 0m0.040s sys 0m0.007s
```

迁移至 `Alpine` 基础镜像

目前，大部分 Docker 官方镜像都已经支持 Alpine 作为基础镜像，可以很容易进行迁移。

例如：

- ubuntu/debian -> alpine
- python:2.7 -> python:2.7-alpine
- ruby:2.3 -> ruby:2.3-alpine

另外，如果使用 `Alpine` 镜像替换 `Ubuntu` 基础镜像，安装软件包时需要用 apk 包管理器替换 apt 工具，如

```
$ apk add --no-cache <package>
```

`Alpine` 中软件安装包的名字可能会与其他发行版有所不同，可以在 `https://pkgs.alpinelinux.org/packages` 网站搜索并确定安装包名称。如果需要的安装包不在主索引内，但是在测试或社区索引中。那么可以按照以下方法使用这些安装包。

```
$ echo "http://dl-4.alpinelinux.org/alpine/edge/testing" >> /etc
/apk/repositories
$ apk --update add --no-cache <package>
```

相关资源

- `Apline` 官网：`http://alpinelinux.org/`
- `Apline` 官方仓库：`https://github.com/alpinelinux`
- `Apline` 官方镜像：`https://hub.docker.com/_/alpine/`
- `Apline` 官方镜像仓库：`https://github.com/gliderlabs/docker-alpine`

Debian/Ubuntu

Debian 和 Ubuntu 都是目前较为流行的 Debian 系的服务器操作系统，十分适合研发场景。Docker Hub 上提供了官方镜像，国内各大容器云服务也基本都提供了相应的支持。

Debian 系统简介



图 *1.23.3.1 - Debian* 操作系统

Debian 是由 GPL 和其他自由软件许可协议授权的自由软件组成的操作系统，由 Debian 计划（Debian Project）组织维护。Debian 计划是一个独立的、分散的组织，由 3000 人志愿者组成，接受世界多个非盈利组织的资金支持，Software in the Public Interest 提供支持并持有商标作为保护机构。Debian 以其坚守 Unix 和自由软件的精神，以及其给予用户的众多选择而闻名。现时 Debian 包括了超过 25,000 个软件包并支持 12 个计算机系统结构。

Debian 作为一个大的系统组织框架，其下有多种不同操作系统核心的分支计划，主要为采用 Linux 核心的 Debian GNU/Linux 系统，其他还有采用 GNU Hurd 核心的 Debian GNU/Hurd 系统、采用 FreeBSD 核心的 Debian GNU/kFreeBSD 系统，以及采用 NetBSD 核心的 Debian GNU/NetBSD 系统。甚至还有利用 Debian 的系统架构和工具，采用 OpenSolaris 核心构建而成的 Nexenta OS 系统。在这些 Debian 系统中，以采用 Linux 核心的 Debian GNU/Linux 最为著名。

众多的 Linux 发行版，例如 Ubuntu、Knoppix 和 Linspire 及 Xandros 等，都基于 Debian GNU/Linux。

使用 **Debian** 官方镜像

读者可以使用 docker search 搜索 Docker Hub，查找 Debian 镜像：

```
$ docker search debian
NAME          DESCRIPTION     STARS     OFFICIAL   AUTOMATED
debian        Debian is...    1565      [OK]
neurodebian   NeuroDebian...    26      [OK]
armbuild/debian port of debian 8                   [OK]
...
```

官方提供了大家熟知的 debian 镜像以及面向科研领域的 neurodebian 镜像。

可以使用 docker run 直接运行 Debian 镜像。

```
$ docker run -it debian bash
root@668e178d8d69:/# cat /etc/issue
Debian GNU/Linux 8
```

Debian 镜像很适合作为基础镜像，构建自定义镜像。

Ubuntu 系统简介



图 *1.23.3.2 - Ubuntu* 操作系统

Ubuntu 是一个以桌面应用为主的GNU/Linux操作系统，其名称来自非洲南部祖鲁语或豪萨语的“ubuntu”一词（官方译名“友帮拓”，另有“吾帮托”、“乌班图”、“有奔头”或“乌斑兔”等译名）。Ubuntu 意思是“人性”以及“我的存在是因为大家的存在”，是非洲传统的一种价值观，类似华人社会的“仁爱”思想。 Ubuntu 基于 Debian 发行版和 GNOME/Unity 桌面环境，与 Debian 的不同在于它每 6 个月会发布一个新版本，每 2 年推出一个长期支持（Long Term Support，LTS）版本，一般支持 3 年时间。

使用 Ubuntu 官方镜像

Ubuntu 相关的镜像有很多，这里使用 `-s 10` 参数，只搜索那些被收藏 10 次以上的镜像。

```
$ docker search -s 10 ubuntu

NAME                                 DESCRIPTION
                     STARS     OFFICIAL   AUTOMATED
ubuntu                               Official Ubuntu base image
                     840       [OK]
dockerfile/ubuntu                    Trusted automated Ubuntu (h
ttp://www.ubunt...   30                   [OK]
crashsystems/gitlab-docker           A trusted, regularly update
d build of GitL...   20                   [OK]
sylvainlasnier/memcached             This is a Memcached 1.4.14
docker images b...   16                   [OK]
ubuntu-upstart                       Upstart is an event-based r
eplacement for ...   16        [OK]
mbentley/ubuntu-django-uwsgi-nginx
                     16                   [OK]
ansible/ubuntu14.04-ansible          Ubuntu 14.04 LTS with ansib
le                   15                   [OK]
clue/ttrss                           The Tiny Tiny RSS feed read
er allows you t...   14                   [OK]
dockerfile/ubuntu-desktop            Trusted automated Ubuntu De
sktop (LXDE) (h...   14                   [OK]
tutum/ubuntu                         Ubuntu image with SSH acces
s. For the root...   12                   [OK]
```

注意，*Docker 1.12* 版本中已经不支持 *--stars* 参数，则可以使用 *-f stars=N* 参数。

根据搜索出来的结果，读者可以自行选择下载镜像并使用。

下面以 ubuntu 14.04 为例，演示如何使用该镜像安装一些常用软件。

首先使用 `-ti` 参数启动容器，登录 bash，查看 ubuntu 的发行版本号。

```
$ docker run -ti ubuntu:14.04 /bin/bash
root@7d93de07bf76:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 14.04.1 LTS
Release:        14.04
Codename:       trusty
```

当试图直接使用 `apt-get` 安装一个软件的时候，会提示 `E: Unable to locate package` 。

```
root@7d93de07bf76:/# apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package curl
```

这并非系统不支持 `apt-get` 命令。Docker 镜像在制作时为了精简清除了 apt 仓库信息，因此需要先执行 `apt-get update` 命令来更新仓库信息。更新信息后即可成功通过 apt-get 命令来安装软件。

```
root@7d93de07bf76:/# apt-get update
Ign http://archive.ubuntu.com trusty InRelease
Ign http://archive.ubuntu.com trusty-updates InRelease
Ign http://archive.ubuntu.com trusty-security InRelease
Ign http://archive.ubuntu.com trusty-proposed InRelease
Get:1 http://archive.ubuntu.com trusty Release.gpg [933 B]
...
```

首先，安装 curl 工具。

```
root@7d93de07bf76:/# apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  ca-certificates krb5-locales libasn1-8-heimdal libcurl3 libgss
api-krb5-2
  libgssapi3-heimdal libhcrypto4-heimdal libheimbase1-heimdal
  libheimntlm0-heimdal libhx509-5-heimdal libidn11 libk5crypto3
libkeyutils1
  libkrb5-26-heimdal libkrb5-3 libkrb5support0 libldap-2.4-2
  libroken18-heimdal librtmp0 libsasl2-2 libsasl2-modules libsas
l2-modules-db
  libwind0-heimdal openssl
...
root@7d93de07bf76:/# curl
curl: try 'curl --help' or 'curl --manual' for more information
```

接下来，再安装 apache 服务。

```
root@7d93de07bf76:/# apt-get install -y apache2
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  apache2-bin apache2-data libapr1 libaprutil1 libaprutil1-dbd-s
qlite3
  libaprutil1-ldap libxml2 sgml-base ssl-cert xml-core
...
```

启动这个 apache 服务，然后使用 curl 来测试本地访问。

```
root@7d93de07bf76:/# service apache2 start
 * Starting web server apache2

                             AH00558: apache2: Could not reliabl
y determine the server's fully qualified domain name, using 172.
17.0.2. Set the 'ServerName' directive globally to suppress this
 message
 *
root@7d93de07bf76:/# curl 127.0.0.1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <!--
    Modified from the Debian original for Ubuntu
    Last updated: 2014-03-19
    See: https://launchpad.net/bugs/1288690
  -->
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=
UTF-8" />
    <title>Apache2 Ubuntu Default Page: It works</title>
    <style type="text/css" media="screen">
...
```

配合使用 `-p` 参数对外映射服务端口，可以允许容器外来访问该服务。

相关资源

- `Debian` 官网：`https://www.debian.org/`
- `Neuro Debian` 官网：`http://neuro.debian.net/`
- `Debian` 官方仓库：`https://github.com/Debian`
- `Debian` 官方镜像：`https://hub.docker.com/_/debian/`
- `Debian` 官方镜像仓库：`https://github.com/tianon/docker-brew-debian/`
- `Ubuntu` 官网：`http://www.ubuntu.org.cn/global`
- `Ubuntu` 官方仓库：`https://github.com/ubuntu`

- `Ubuntu` 官方镜像：`https://hub.docker.com/_/ubuntu/`
- `Ubuntu` 官方镜像仓库：`https://github.com/tianon/docker-brew-ubuntu-core`

CentOS/Fedora

CentOS 系统简介

CentOS 和 Fedora 都是基于 Redhat 的常见 Linux 分支。CentOS 是目前企业级服务器的常用操作系统；Fedora 则主要面向个人桌面用户。



图 *1.23.4.1 - CentOS* 操作系统

CentOS（Community Enterprise Operating System，中文意思是：社区企业操作系统），它是基于 Red Hat Enterprise Linux 源代码编译而成。由于 CentOS 与 Redhat Linux 源于相同的代码基础，所以很多成本敏感且需要高稳定性的公司就使用 CentOS 来替代商业版 Red Hat Enterprise Linux。CentOS 自身不包含闭源软件。

使用 CentOS 官方镜像

首先使用 `docker search` 命令来搜索标星至少为 25 的 CentOS 相关镜像。

```
$ docker search -f stars=25 centos
NAME        DESCRIPTION      STARS     OFFICIAL   AUTOMATED
centos      The official...  2543      [OK]
jdeathe/centos-ssh           27                   [OK]
```

使用 docker run 直接运行最新的 CentOS 镜像，并登录 bash。

```
$ docker run -it centos bash
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
3d8673bd162a: Pull complete
Digest: sha256:a66ffcb73930584413de83311ca11a4cb4938c9b2521d3310
26dad970c19adf4
Status: Downloaded newer image for centos:latest
[root@43eb3b194d48 /]# cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core)
```

Fedora 系统简介



图 *1.23.4.2 - Fedora* 操作系统

Fedora 由 Fedora Project 社区开发，红帽公司赞助的 Linux 发行版。它的目标是创建一套新颖、多功能并且自由和开源的操作系统。Fedora 的功能对于用户而言，它是一套功能完备的，可以更新的免费操作系统，而对赞助商 Red Hat 而言，它是许多新技术的测试平台。被认为可用的技术最终会加入到 Red Hat Enterprise Linux中。

使用 Fedora 官方镜像

首先使用 `docker search` 命令来搜索标星至少为 2 的 Fedora 相关镜像，结果如下。

```
$ docker search -f stars=2 fedora
NAME                         DESCRIPTION
         STARS     OFFICIAL    AUTOMATED
fedora                       Official Docker builds of Fedora
         433       [OK]
dockingbay/fedora-rust       Trusted build of Rust programming langu
age...   3                          [OK]
gluster/gluster-fedora       Official GlusterFS image [ Fedora 21 +
Glu...   3                          [OK]
startx/fedora                Simple container used for all startx ba
sed...   2                          [OK]
```

使用 docker run 命令直接运行 Fedora 官方镜像，并登录 bash。

```
$ docker run -it fedora bash
Unable to find image 'fedora:latest' locally
latest: Pulling from library/fedora
2bf01635e2a0: Pull complete
Digest: sha256:64a02df6aac27d1200c2572fe4b9949f1970d05f74d367ce4
af994ba5dc3669e
Status: Downloaded newer image for fedora:latest
[root@196ca341419b /]# cat /etc/redhat-release
Fedora release 24 (Twenty Four)
```

相关资源

- `Fedora` 官网： `https://getfedora.org/`
- `Fedora` 官方仓库： `https://github.com/fedora-infra`
- `Fedora` 官方镜像： `https://hub.docker.com/_/fedora/`
- `Fedora` 官方镜像仓库： `https://github.com/fedora-cloud/docker-brew-fedora`
- `CentOS` 官网： `https://getfedora.org/`
- `CentOS` 官方仓库： `https://github.com/CentOS`
- `CentOS` 官方镜像： `https://hub.docker.com/_/centos/`
- `CentOS` 官方镜像仓库： `https://github.com/CentOS/CentOS-Dockerfiles`

CoreOS

简介



图 *1.23.5.1 - CoreOS -* 容器化操作系统

CoreOS 是一个轻量级容器化 Linux 发行版，专为数据中心场景而设计，旨在通过轻量的系统架构和灵活的应用程序部署能力简化数据中心的维护成本和复杂度。CoreOS 作为 Docker 生态圈中的重要一员，日益得到各大云服务商的重视。CoreOS 于 2014 年 7 月首次发布了稳定版本。

与其他历史悠久、使用广泛的 Linux 操作系统相比，CoreOS 拥有下面几个优点。

- CoreOS 通过容器化（Containerized）的运算环境向应用程序提供运算资源。

传统类 Unix 系统往往提供包管理工具。随着系统安装更多的程序，而程序的依赖要求各不相同，这就容易出现“依赖地狱”（Dependency Hell），系统的更新和安装会非常痛苦。

CoreOS 的应用程序通过 Docker 运行在容器中，彼此之间共享系统内核和资源，同时互不可见。这种方式使得操作系统、应用程序及运行环境之间的耦合度大大降低。相对于传统的部署方式而言，运维可以更加灵活便捷的在 CoreOS 集群中部署应用程序，同时各运行环境之间的干扰更少，操作系统自身的维护也更加容易。

- CoreOS 采用双系统分区（Dual Root Partition）设计。

CoreOs 的两个分区在系统运行期间各司其职，它们分别被设置成主动模式和被动模式。主动分区负责系统运行，被动分区负责系统升级。一旦 CoreOS 发布了新版系统，运维只需下载一个完整的系统安装文件至被动分区，然后设置系统下一次重启时从新版本分区启动，将主、被动分区的职能进行调换即可。在 CoreOS 系统运行期间，系统分区被设置成只读状态，这样也确保了 CoreOS 的安全性。CoreOS 的升级过程在默认条件下将自动完成，并且通过 cgroup 对升级过程中使用到的网络和磁盘资源进行限制，这将系统升级所带来的影响降至最低。

- CoreOS 使用 Systemd 取代 SysV 作为系统和服务的管理工具。

与 SysV 相比，Systemd 不仅可以更好的追踪系统进程，还具备优秀的并行化处理能力。Systemd 将自己的按需启动的特性与 Docker 的快速启动能力相结合，在 CoreOS 集群中大规模部署 Docker Containers 的业务中优势明显。Systemd 引入了“target”的概念，每个 target 应用于一个特定的服务，并且可以通过继承一个已有的 target 来扩展额外的功能，即操作系统对系统上运行的服务拥有更好的控制力。

CoreOS 团队还推出了很多有益的工具，包括 etcd, fleet, flannel 等。

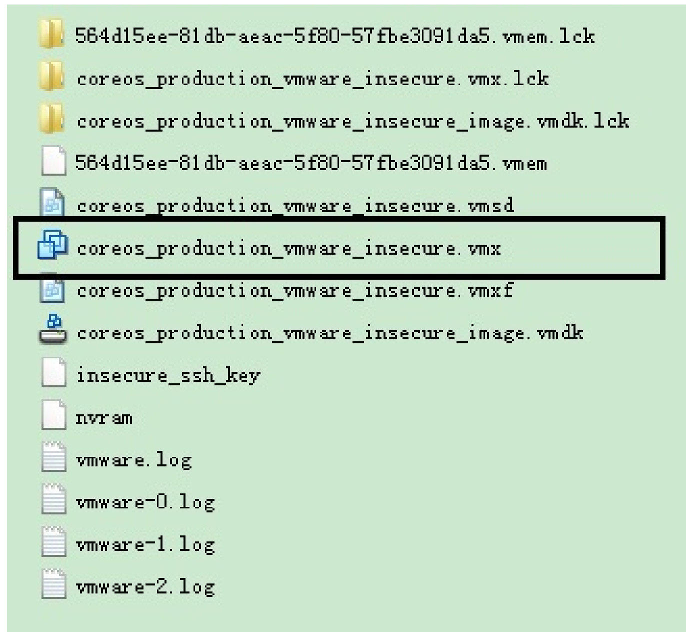
安装 CoreOS 有几种方法，笔者推荐初学者使用 VMware Workstation 虚拟机方式来运行 CoreOS。VMware Workstation 工具则可以前往其官网 `http://www.vmware.com/products/workstation/` 下载获取。

使用官方镜像

获取虚拟机镜像

从官方网站下载 CoreOS 镜像，地址为 https:\/\/coreos.com\/releases\/ 。

如果读者已经安装 VMware Workstation，则解压镜像包后双击 `vmx` 文件：



!

双击 `vmx` 文件后，即可启动 CoreOS 虚拟机，如下图所示：

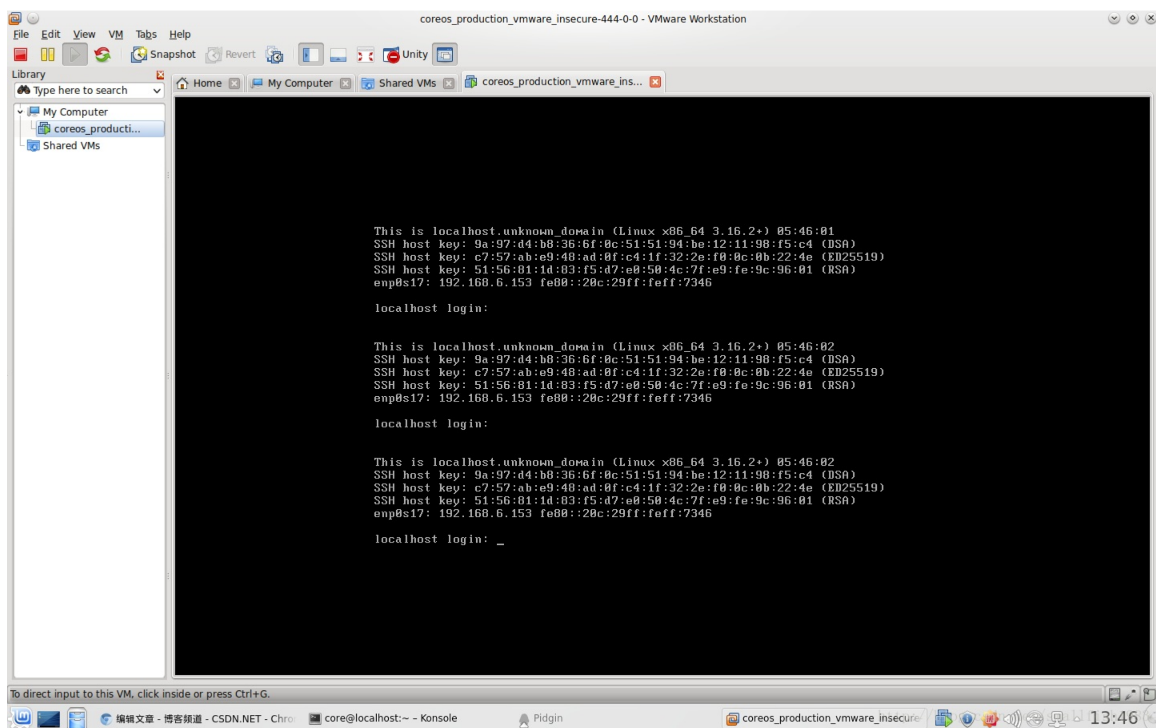


图 1.23.5.2 - 启动 CoreOS 虚拟机

注意：使用免费版 *VMware Player* 运行 *CoreOS* 官方镜像时，可能出现无法通过 *DHCP* 自动获取 *IP* 地址的问题，读者可配置静态地址或使用 *VMware Workstation* 来运行 `vmx` 文件。

获取地址信息

此时CoreOS 系统已经在 VMware Station 中启动，显示登录提示：

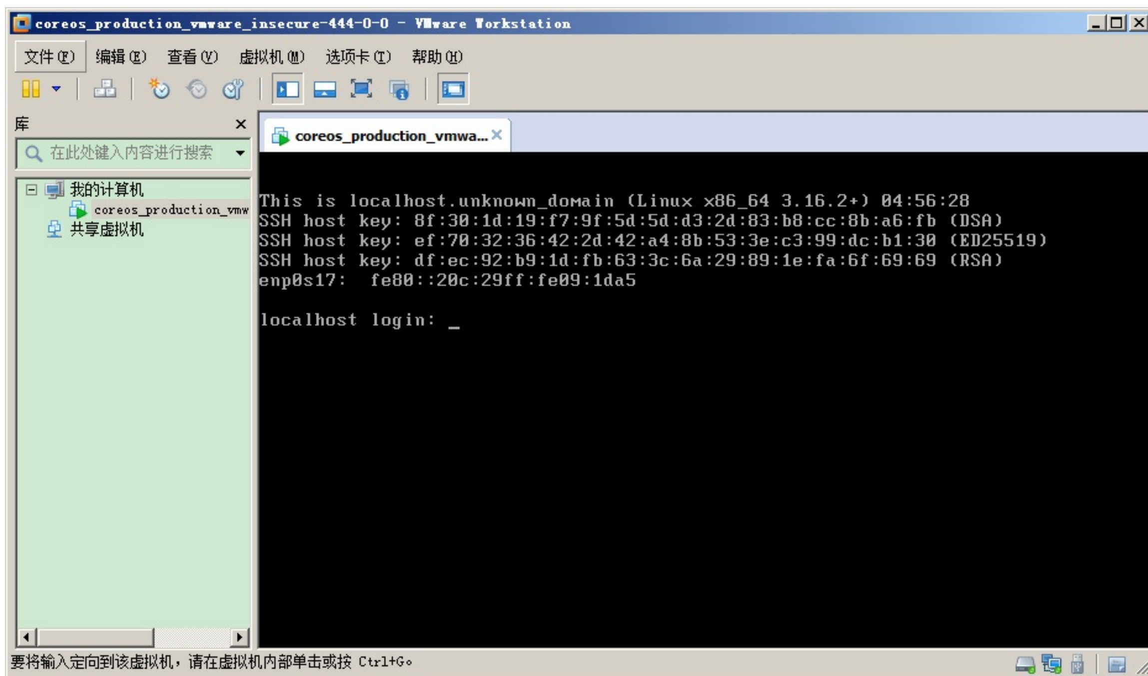


图 *1.23.5.3* - 登录 *CoreOS*

直接按回车键，获取当前系统的 IP 地址，如下图所示：

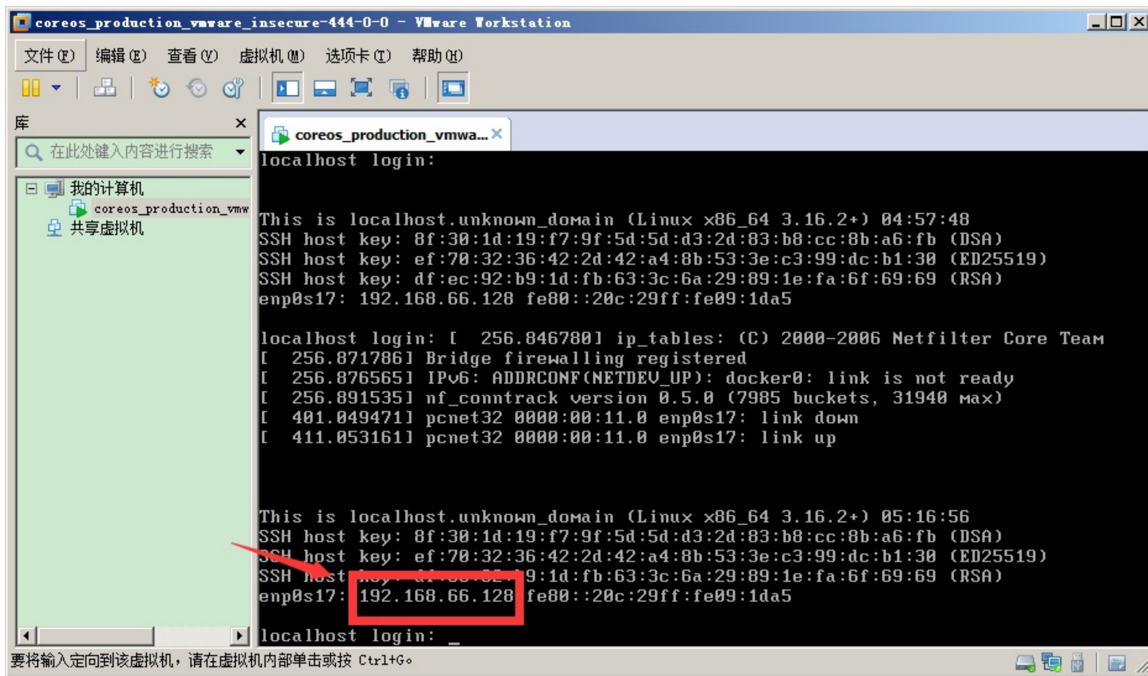


图 *1.23.5.4 - 获取 IP 地址*

如上图所示，查看此时 CoreOS 的 IP地址是： `192.168.66.128` 。

使用 SSH 客户端访问镜像

笔者以 Windows 环境为例，使用 SecureCRT 工具进行连接。此处读者需要确定：

- CoreOS 虚拟机的 IP地址
- CoreOS 虚拟机的文件目录下含有 `insecure_ssh_key` 公钥文件

打开 SecureCRT，建立新的 SSH 连接，如下图示：

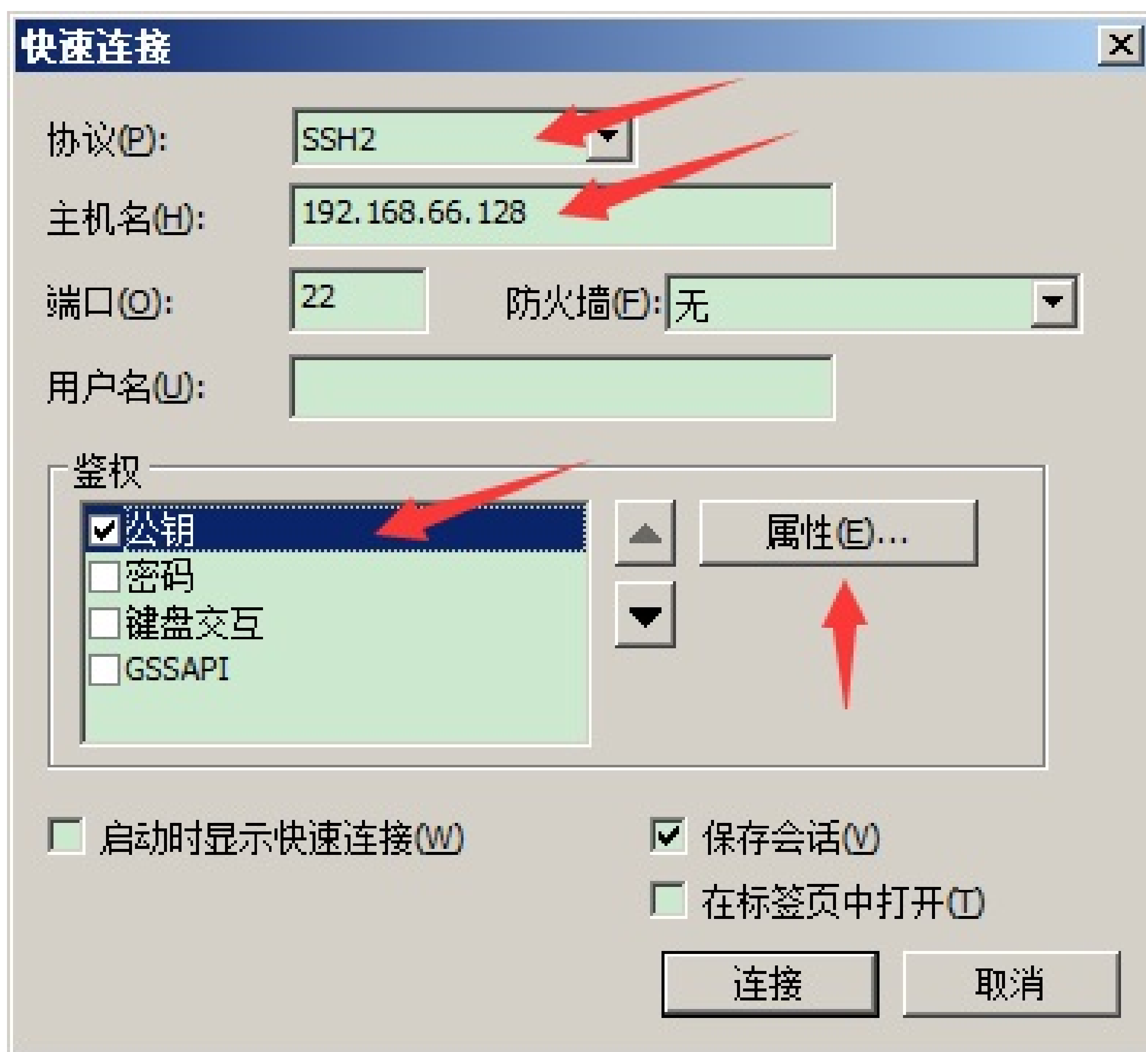


图 *1.23.5.5* - 使用 *SecureCRT* 访问 *CoreOS*

点击 `属性` 按钮添加 `insecure_ssh_key` 公钥文件后，即可点击 `连接` 。

如果连接成功，则读者可以看到命令行页面，读者在命令行中查看 Docker 的版本信息：

```
$ docker version
```

结果如下图示：

```
CoreOS (alpha)
core@localhost ~ $ docker version
Client version: 1.2.0
Client API version: 1.14
Go version (client): go1.3.1
Git commit (client): fa7b24f
OS/Arch (client): linux/amd64
Server version: 1.2.0
Server API version: 1.14
Go version (server): go1.3.1
Git commit (server): fa7b24f
core@localhost ~ $
```

图 *1.23.5.6* - 查看 *Docker* 版本信息

此时，CoreOS 虚拟机已经成功运行，并且读者可以使用 SSH 客户端方便的操作 CoreOS 虚拟机。 Docker 已经内置于 CoreOS 中，读者可以进行各种 Docker 操作，如下图示：

192.168.66.128 - SecureCRT

文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)

192.168.66.128

```
core@localhost ~ $ docker pull php
Pulling repository php
6b3a85e4a74a: Pulling image (5.3.29-cli) from php, endpoint: https://registry-1.
docker.io/v1/
6b0d47f47ad7: Pulling image (5.5.18-apache) from php, endpoint: https://registry
-1.docker.io/v1/
a7fd2d55bb86: Pulling image (5.5.16-apache) from php, endpoint: https://registry
-1.docker.io/v1/
f021d64f9a8c: Pulling image (5.5-apache) from php, endpoint: https://registry-1.
docker.io/v1/
29e51d0556bc: Pulling image (5.4.32-apache) from php, endpoint: https://registry
-1.docker.io/v1/
9a7aa409f758: Pulling image (5.6.3-apache) from php, endpoint: https://registry-
1.docker.io/v1/
bbf02dfe667b: Pulling image (5.4.33-apache) from php, endpoint: https://registry
-1.docker.io/v1/
c3f5a4023aba: Pulling image (5.6.0-apache) from php, endpoint: https://registry-
9a7aa409f758: Pulling dependent layers
bbf02dfe667b: Pulling dependent layers
a7fd2d55bb86: Pulling dependent layers
9d96aa25e739: Pulling dependent layers
8314ed40214d: Pulling dependent layers
a5efe7a0d01b: Pulling dependent layers
0383dbddafe9: Pulling image (5.4.32-cli) from php, endpoint: https://registry-1.
256ac99db932: Pulling dependent layers
00fa73ec2793: Pulling dependent layers
e4603badbed5: Pulling image (5.6.2-apache) from php, endpoint: https://registry-
1c6588912a40: Pulling dependent layers
5431dbd92412: Pulling image (5.4.35-apache) from php, endpoint: https://registry
dcdd5b107bc2: Pulling dependent layers
0ba82ab21309: Pulling image (5.6-fpm) from php, endpoint: https://registry-1.doc
42ac699b8550: Pulling dependent layers
630aabff79c5: Pulling dependent layers
44ededfd6597: Pulling dependent layers
b99ea5c97254: Pulling dependent layers
68fec2bed78d: Pulling dependent layers
d4109f67485c: Pulling dependent layers
2bdbadf3469b: Pulling dependent layers
802b1aaa89da: Pulling dependent layers
7a0d03c7b9dc: Pulling dependent layers
900ad5fcb607: Pulling dependent layers
511136ea3c5a: Pulling metadata
```

图 *1.23.5.7 - 使用 Docker 命令*

如果读者的本机环境是 Linux 系统，读者可以使用 SSH 公钥（在解压后的根目录下），直接使用 `ssh` 命令连接 CoreOS 虚拟机，并使用 `ip a` 命令查看 IP 地址信息。

如下所示：

```
$ ssh -i ~/insecure_ssh_key core@192.168.6.153
CoreOS (alpha)
core@localhost ~ $ ls
core@localhost ~ $ docker ps
CONTAINER ID        IMAGE                   COMMAND                CREA
TED              STATUS                 PORTS                 NAMES
core@localhost ~ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKN
OWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfi
fo_fast state UNKNOWN qlen 1000
    link/ether 00:0c:29:ff:73:46 brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.153/20 brd 192.168.15.255 scope global dynami
c enp0s17
       valid_lft 604500sec preferred_lft 604500sec
    inet6 fe80::20c:29ff:feff:7346/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc n
oqueue state DOWN
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
core@localhost ~ $
```

相关资源

- `CoreOS` 官网：`https://coreos.com/`
- `CoreOS` 官方仓库：`https://github.com/coreos/`

本章小结

本章讲解了典型操作系统镜像的下载和使用。

除了官方的镜像外，在 DockerHub 上还有许多第三方组织或个人上传的 Docker 镜像。

读者可以根据具体情况来选择。一般来说：

- 官方镜像体积都比较小，只带有一些基本的组件。 精简的系统有利于安全、稳定和高效的运行，也适合进行定制化。
- 个别第三方（如 tutum，已被 Docker 收购）镜像质量也非常高。这些镜像通常针对某个具体应用进行配置，比如：包含 LAMP 组件的 Ubuntu 镜像。
- 处于安全考虑，几乎所有官方制作的镜像都没有安装 SSH 服务，无法使用用户名和密码直接登录。

后续章节中，笔者将介绍如何创建一个带 SSH 服务的 Docker 镜像。

附录

常见问题总结

镜像相关

如何批量清理临时镜像文件

答：可以使用 `docker rmi $(docker images -q -f dangling=true)` 命令。

如何查看镜像支持的环境变量？

答：可以使用 `docker run IMAGE env` 命令。

本地的镜像文件都存放在哪里？

答：与 Docker 相关的本地资源都存放在 `/var/lib/docker/` 目录下，以 aufs 文件系统为例，其中 container 目录存放容器信息，graph 目录存放镜像信息，aufs 目录下存放具体的镜像层文件。

构建 Docker 镜像应该遵循哪些原则？

答：整体原则上，尽量保持镜像功能的明确和内容的精简，要点包括

- 尽量选取满足需求但较小的基础系统镜像，例如大部分时候可以选择 debian:wheezy 或 debian:jessie 镜像，仅有不足百兆大小;
- 清理编译生成文件、安装包的缓存等临时文件；
- 安装各个软件时候要指定准确的版本号，并避免引入不需要的依赖；
- 从安全角度考虑，应用要尽量使用系统的库和依赖；
- 如果安装应用时候需要配置一些特殊的环境变量，在安装后要还原不需要保持的变量值；
- 使用 Dockerfile 创建镜像时候要添加 .dockerignore 文件或使用干净的工作目录。

碰到网络问题，无法 pull 镜像，命令行指定 http_proxy 无效？

答：在 Docker 配置文件中添加 `export http_proxy="http://<PROXY_HOST>:<PROXY_PORT>"` ，之后重启 Docker 服务即可。

容器相关

容器退出后，通过 docker ps 命令查看不到，数据会丢失么？

答：容器退出后会处于终止（exited）状态，此时可以通过 `docker ps -a` 查看。其中的数据也不会丢失，还可以通过 `docker start` 命令来启动它。只有删除掉容器才会清除所有数据。

如何停止所有正在运行的容器

答：可以使用 `docker kill $(docker ps -q)` 命令。

如何清理批量后台停止的容器？

答：可以使用 `docker rm -f $(docker ps -qa)` 命令。

如何获取某个容器的 PID 信息？

答：可以使用 `docker inspect --format '{{ .State.Pid }}' <CONTAINER ID or NAME>` 命令。

如何获取某个容器的 IP 地址？

答：可以使用 `docker inspect --format '{{ .NetworkSettings.IPAddress }}' <CONTAINER ID or NAME>` 命令

如何给容器指定一个固定 IP 地址，而不是每次重启容器 IP 地址都会变？

答：目前 Docker 并没有提供直接的对容器 IP 地址的管理支持，用户可以考虑参考第三部分中高级网络配置章节中介绍的创建点对点连接例子，来手动配置容器的静态 IP。或者在启动容器后，再手动进行修改（可参考其它类的问题“如何进入 *Docker* 容器的网络命名空间？”）。

如何临时退出一个正在交互的容器的终端，而不终止它？

答：按 `Ctrl-p Ctrl-q`。如果按 `Ctrl-c` 往往会让容器内应用进程终止，进而会终止容器。

使用 `docker port` 命令映射容器的端口时，系统报错“Error: No public port '80' published for xxx”？

答：

- 创建镜像时 Dockerfile 要通过 EXPOSE 指定正确的开放端口；
- 容器启动时指定 `PublishAllPort = true`。

可以在一个容器中同时运行多个应用进程么？

答：一般并不推荐在同一个容器内运行多个应用进程。如果有类似需求，可以通过一些额外的进程管理机制，比如 supervisord 来管理所运行的进程。可以参考 https://docs.docker.com/articles/using_supervisord/。

如何控制容器占用系统资源（CPU、内存）的份额？

答：在使用 `docker create` 命令创建容器或使用 `docker run` 创建并启动容器的时候，可以使用 -c|--cpu-shares[=0] 参数来调整容器使用 CPU 的权重；使用 -m|--memory[=MEMORY] 参数来调整容器使用内存的大小。

仓库相关

仓库（Repository）、注册服务器（Registry）、注册索引（Index）有何关系？

首先，仓库是存放一组关联镜像的集合，比如同一个应用的不同版本的镜像。

注册服务器是存放实际的镜像文件的地方。注册索引则负责维护用户的账号、权限、搜索、标签等的管理。因此，注册服务器利用注册索引来实现认证等管理。

从非官方仓库（例如 non-official-repo.com）下载镜像时候，有时候会提示“Error: Invalid registry endpoint https://non-official-repo.com/v1/……”？

答：Docker 自 1.3.0 版本往后，加强了对镜像安全性的验证，需要添加私有仓库证书，或者手动添加对非官方仓库的信任。 编辑 Docker 配置文件，在其中添加：

```
DOCKER_OPTS="--insecure-registry non-official-repo"
```

之后，重启 Docker 服务即可。

配置相关

Docker 的配置文件放在哪里，如何修改配置？

答：使用 upstart 的系统（如Ubuntu 14.04）的配置文件在 `/etc/default/docker` ，使用systemd 的系统（如 Ubuntu 16.04、Centos 等）的配置文件在 `/etc/systemd/system/docker.service.d/docker.conf` 。

Ubuntu 下面的配置文件内容如下，读者可以参考配。（如果出现该文件不存在的情况，重启或者自己新建一个文件都可以解决。）

```
# Customize location of Docker binary (especially for developmen
t testing).
#DOCKERD="/usr/local/bin/dockerd"

# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"

# If you need Docker to use an HTTP proxy, it can also be specif
ied here.
#export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary f
iles go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

如何更改 Docker 的默认存储位置？

答：Docker 的默认存储位置是 `/var/lib/docker`，如果希望将 Docker 的本地文件存储到其他分区，可以使用 Linux 软连接的方式来完成，或者在启动 daemon 时通过 `-g` 参数指定。

例如，如下操作将默认存储位置迁移到 /storage/docker。

```
[root@s26 ~]# df -h
Filesystem                      Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup-lv_root     50G  5.3G   42G  12% /
tmpfs                            48G  228K   48G   1% /dev/shm
/dev/sda1                       485M   40M  420M   9% /boot
/dev/mapper/VolGroup-lv_home    222G  188M  210G   1% /home
/dev/sdb2                       2.7T  323G  2.3T  13% /storage
[root@s26 ~]# service docker stop
[root@s26 ~]# cd /var/lib/
[root@s26 lib]# mv docker /storage/
[root@s26 lib]# ln -s /storage/docker/ docker
[root@s26 lib]# ls -la docker
lrwxrwxrwx. 1 root root 15 11月 17 13:43 docker -> /storage/dock
er
[root@s26 lib]# service docker start
```

使用内存和 swap 限制启动容器时候报警告："WARNING: Your kernel does not support cgroup swap limit. WARNING: Your kernel does not support swap limit capabilities. Limitation discarded."？

答：这是因为系统默认没有开启对内存和 swap 使用的统计功能，引入该功能会带来性能的下降。要开启该功能，可以采取如下操作：

- 编辑 `/etc/default/grub` 文件（Ubuntu 系统为例），配置 `GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"`；
- 更新 grub：`$ sudo update-grub`；
- 重启系统，即可。

Docker 与虚拟化

Docker 与 LXC（Linux Container）有何不同？

答：LXC 利用 Linux 上相关技术实现了容器。Docker 则在如下的几个方面进行了改进：

- 移植性：通过抽象容器配置，容器可以实现从一个平台移植到另一个平台；
- 镜像系统：基于 AUFS 的镜像系统为容器的分发带来了很多的便利，同时共同的镜像层只需要存储一份，实现高效率的存储；
- 版本管理：类似于Git的版本管理理念，用户可以更方便的创建、管理镜像文件；
- 仓库系统：仓库系统大大降低了镜像的分发和管理的成本；
- 周边工具：各种现有工具（配置管理、云平台）对 Docker 的支持，以及基于 Docker的 PaaS、CI 等系统，让 Docker 的应用更加方便和多样化。

Docker 与 Vagrant 有何不同？

答：两者的定位完全不同。

- Vagrant 类似 Boot2Docker（一款运行 Docker 的最小内核），是一套虚拟机的管理环境。Vagrant 可以在多种系统上和虚拟机软件中运行，可以在 Windows，Mac 等非 Linux 平台上为 Docker 提供支持，自身具有较好的包装性和移植性。
- 原生的 Docker 自身只能运行在 Linux 平台上，但启动和运行的性能都比虚拟机要快，往往更适合快速开发和部署应用的场景。

简单说：Vagrant 适合用来管理虚拟机，而 Docker 适合用来管理应用环境。

开发环境中 Docker 和 Vagrant 该如何选择？

答： Docker 不是虚拟机，而是进程隔离，对于资源的消耗很少，但是目前需要 Linux 环境支持。Vagrant 是虚拟机上做的封装，虚拟机本身会消耗资源。

如果本地使用的 Linux 环境，推荐都使用 Docker。

如果本地使用的是 OSX 或者 Windows 环境，那就需要开虚拟机，单一开发环境下 Vagrant 更简单；多环境开发下推荐在 Vagrant 里面再使用 Docker 进行环境隔离。

其它

Docker 能在非 Linux 平台（比如 Windows 或 MacOS）上运行么？

答：可以。目前需要使用 docker for mac、boot2docker 等软件创建一个轻量级的 Linux 虚拟机层。

如何将一台宿主主机的 docker 环境迁移到另外一台宿主主机

答：停止 Docker 服务。将整个 Docker 存储文件夹复制到另外一台宿主主机，然后调整另外一台宿主主机的配置即可。

如何进入 Docker 容器的网络命名空间？

答：Docker 在创建容器后，删除了宿主主机上 `/var/run/netns` 目录中的相关的网络命名空间文件。因此，在宿主主机上是无法看到或访问容器的网络命名空间的。

用户可以通过如下方法来手动恢复它。

首先，使用下面的命令查看容器进程信息，比如这里的1234。

```
$ docker inspect --format='{{. State.Pid}} ' $container_id
1234
```

接下来，在 /proc 目录下，把对应的网络命名空间文件链接到 /var/run/netns 目录。

```
$ sudo ln -s /proc/1234/ns/net /var/run/netns/
```

然后，在宿主主机上就可以看到容器的网络命名空间信息。例如

```
$ sudo ip netns show
1234
```

此时，用户可以通过正常的系统命令来查看或操作容器的命名空间了。例如修改容器的 IP 地址信息为 `172.17.0.100/16` 。

```
$ sudo ip netns exec 1234 ifconfig eth0 172.17.0.100/16
```

如何获取容器绑定到本地那个 veth 接口上？

答：Docker 容器启动后，会通过 veth 接口对连接到本地网桥，veth 接口命名跟容器命名毫无关系，十分难以找到对应关系。

最简单的一种方式是通过查看接口的索引号，在容器中执行 `ip a` 命令，查看到本地接口最前面的接口索引号，如 `205` ，将此值加上 1，即 `206` ，然后在本地主机执行 `ip a` 命令，查找接口索引号为 `206` 的接口，两者即为连接的 veth 接口对。

热门镜像介绍

本章将介绍一些热门镜像的功能，使用方法和生成它们的 Dockerfile 等。包括 Ubuntu、CentOS、MySQL、MongoDB、Redis、Nginx、Wordpress、Node.js 等。

Ubuntu

基本信息

Ubuntu 是流行的 Linux 发行版，其自带软件版本往往较新一些。

该仓库位于 https://hub.docker.com/_/ubuntu/ ，提供了 Ubuntu 从 12.04 ~ 16.10 各个版本的镜像。

使用方法

默认会启动一个最小化的 Ubuntu 环境。

```
$ docker run --name some-ubuntu -i -t ubuntu
root@523c70904d54:/#
```

Dockerfile

12.04 版本

```
FROM scratch
ADD precise-core-amd64.tar.gz /

# a few minor docker-specific tweaks
# see https://github.com/dotcloud/docker/blob/master/contrib/mki
mage/debootstrap
RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
    && echo 'exit 101' >> /usr/sbin/policy-rc.d \
    && chmod +x /usr/sbin/policy-rc.d \
    \
    && dpkg-divert --local --rename --add /sbin/initctl \
    && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
    && sed -i 's/^exit.*/exit 0/' /sbin/initctl \
    \
    && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-
speedup \
```

```
    \
    && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/
*.deb /var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin
 || true"; };' > /etc/apt/apt.conf.d/docker-clean \
    && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/ar
chives/*.deb /var/cache/apt/archives/partial/*.deb /var/cache/ap
t/*.bin || true"; };' >> /etc/apt/apt.conf.d/docker-clean \
    && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache ""
;' >> /etc/apt/apt.conf.d/docker-clean \
    \
    && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/d
ocker-no-languages \
    \
    && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTy
pes::Order:: "gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes

# delete all the apt list files since they're big and get stale
quickly
RUN rm -rf /var/lib/apt/lists/*
# this forces "apt-get update" in dependent images, which is als
o good

# enable the universe
RUN sed -i 's/^#\s*\(deb.*universe\)$/\1/g' /etc/apt/sources.lis
t

# upgrade packages for now, since the tarballs aren't updated fr
equently enough
RUN apt-get update && apt-get dist-upgrade -y && rm -rf /var/lib
/apt/lists/*

# overwrite this with 'CMD []' in a dependent Dockerfile
CMD ["/bin/bash"]
```

14.04 版本

```
FROM scratch
ADD trusty-core-amd64.tar.gz /
```

```
# a few minor docker-specific tweaks
# see https://github.com/dotcloud/docker/blob/master/contrib/mki
mage/debootstrap
RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
    && echo 'exit 101' >> /usr/sbin/policy-rc.d \
    && chmod +x /usr/sbin/policy-rc.d \
    \
    && dpkg-divert --local --rename --add /sbin/initctl \
    && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
    && sed -i 's/^exit.*/exit 0/' /sbin/initctl \
    \
    && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-
speedup \
    \
    && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/
*.deb /var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin
 || true"; };' > /etc/apt/apt.conf.d/docker-clean \
    && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/ar
chives/*.deb /var/cache/apt/archives/partial/*.deb /var/cache/ap
t/*.bin || true"; };' >> /etc/apt/apt.conf.d/docker-clean \
    && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache ""
;' >> /etc/apt/apt.conf.d/docker-clean \
    \
    && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/d
ocker-no-languages \
    \
    && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTy
pes::Order:: "gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes

# delete all the apt list files since they're big and get stale
quickly
RUN rm -rf /var/lib/apt/lists/*
# this forces "apt-get update" in dependent images, which is als
o good

# enable the universe
RUN sed -i 's/^#\s*\(deb.*universe\)$/\1/g' /etc/apt/sources.lis
t

# upgrade packages for now, since the tarballs aren't updated fr
```

```
equently enough
RUN apt-get update && apt-get dist-upgrade -y && rm -rf /var/lib
/apt/lists/*

# overwrite this with 'CMD []' in a dependent Dockerfile
CMD ["/bin/bash"]
```

16.04 版本

```
FROM scratch
ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /

# a few minor docker-specific tweaks
# see https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8ee
c0740716226fadb3735c/contrib/mkimage/debootstrap
RUN set -xe \
    \
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec074
0716226fadb3735c/contrib/mkimage/debootstrap#L40-L48
    && echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
    && echo 'exit 101' >> /usr/sbin/policy-rc.d \
    && chmod +x /usr/sbin/policy-rc.d \
    \
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec074
0716226fadb3735c/contrib/mkimage/debootstrap#L54-L56
    && dpkg-divert --local --rename --add /sbin/initctl \
    && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
    && sed -i 's/^exit.*/exit 0/' /sbin/initctl \
    \
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec074
0716226fadb3735c/contrib/mkimage/debootstrap#L71-L78
    && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-
speedup \
    \
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec074
0716226fadb3735c/contrib/mkimage/debootstrap#L85-L105
    && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/
*.deb /var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin
 || true"; };' > /etc/apt/apt.conf.d/docker-clean \
```

```
    && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/ar
chives/*.deb /var/cache/apt/archives/partial/*.deb /var/cache/ap
t/*.bin || true"; };' >> /etc/apt/apt.conf.d/docker-clean \
    && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache ""
;' >> /etc/apt/apt.conf.d/docker-clean \
    \
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec074
0716226fadb3735c/contrib/mkimage/debootstrap#L109-L115
    && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/d
ocker-no-languages \
    \
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec074
0716226fadb3735c/contrib/mkimage/debootstrap#L118-L130
    && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTy
pes::Order:: "gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes \
    \
# https://github.com/docker/docker/blob/9a9fc01af8fb5d98b8eec074
0716226fadb3735c/contrib/mkimage/debootstrap#L134-L151
    && echo 'Apt::AutoRemove::SuggestsImportant "false";' > /etc
/apt/apt.conf.d/docker-autoremove-suggests

# delete all the apt list files since they're big and get stale
quickly
RUN rm -rf /var/lib/apt/lists/*
# this forces "apt-get update" in dependent images, which is als
o good

# enable the universe
RUN sed -i 's/^#\s*\(deb.*universe\)$/\1/g' /etc/apt/sources.lis
t

# overwrite this with 'CMD []' in a dependent Dockerfile
CMD ["/bin/bash"]
```

CentOS

基本信息

CentOS 是流行的 Linux 发行版，其软件包大多跟 RedHat 系列保持一致。

该仓库位于 https://hub.docker.com/_/centos/ ，提供了 CentOS 从 5 ~ 7 各个版本的镜像。

使用方法

默认会启动一个最小化的 CentOS 环境。

```
$ docker run --name some-centos -i -t centos bash
bash-4.2#
```

Dockerfile

CentOS 5 版本

```
FROM scratch
MAINTAINER The CentOS Project <cloud-ops@centos.org>
ADD c5-docker.tar.xz /
LABEL name="CentOS Base Image" \
    vendor="CentOS" \
    license="GPLv2" \
    build-date="2016-03-31"

# Default command
CMD ["/bin/bash"]
```

CentOS 6 版本

```
FROM scratch
MAINTAINER https://github.com/CentOS/sig-cloud-instance-images
ADD centos-6-docker.tar.xz /

LABEL name="CentOS Base Image" \
    vendor="CentOS" \
    license="GPLv2" \
    build-date="20160729"

CMD ["/bin/bash"]
```

CentOS 7 版本

```
FROM scratch
MAINTAINER https://github.com/CentOS/sig-cloud-instance-images
ADD centos-7-docker.tar.xz /

LABEL name="CentOS Base Image" \
    vendor="CentOS" \
    license="GPLv2" \
    build-date="20160729"

CMD ["/bin/bash"]
```

MySQL

基本信息

MySQL 是开源的关系数据库实现。

该仓库位于 https://hub.docker.com/_/mysql/ ，提供了 MySQL 各个版本的镜像，包括 5.6 系列、5.7 系列等。

使用方法

默认会在 `3306` 端口启动数据库。

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpa
ssword -d mysql
```

之后就可以使用其它应用来连接到该容器。

```
$ docker run --name some-app --link some-mysql:mysql -d applicat
ion-that-uses-mysql
```

或者通过 `mysql` 。

```
$ docker run -it --link some-mysql:mysql --rm mysql sh -c 'exec
mysql -h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT
" -uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
```

Dockerfile

5.6 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assign
```

```
ed consistently, regardless of whatever dependencies get added
RUN groupadd -r mysql && useradd -r -g mysql mysql

# FATAL ERROR: please install the following Perl modules before
executing /usr/local/mysql/scripts/mysql_install_db:
# File::Basename
# File::Copy
# Sys::Hostname
# Data::Dumper
RUN apt-get update && apt-get install -y perl --no-install-recom
mends && rm -rf /var/lib/apt/lists/*

# mysqld: error while loading shared libraries: libaio.so.1: can
not open shared object file: No such file or directory
RUN apt-get update && apt-get install -y libaio1 && rm -rf /var/
lib/apt/lists/*

# gpg: key 5072E1F5: public key "MySQL Release Engineering <mysq
l-build@oss.oracle.com>" imported
RUN gpg --keyserver pgp.mit.edu --recv-keys A4A9406876FCBD3C4567
70C88C718D3B5072E1F5

ENV MYSQL_MAJOR 5.6
ENV MYSQL_VERSION 5.6.20

# note: we're pulling the *.asc file from mysql.he.net instead o
f dev.mysql.com because the official mirror 404s that file for w
hatever reason - maybe it's at a different path?
RUN apt-get update && apt-get install -y curl --no-install-recom
mends && rm -rf /var/lib/apt/lists/* \
    && curl -SL "http://dev.mysql.com/get/Downloads/MySQL-$MYSQL
_MAJOR/mysql-$MYSQL_VERSION-linux-glibc2.5-x86_64.tar.gz" -o mys
ql.tar.gz \
    && curl -SL "http://mysql.he.net/Downloads/MySQL-$MYSQL_MAJO
R/mysql-$MYSQL_VERSION-linux-glibc2.5-x86_64.tar.gz.asc" -o mysq
l.tar.gz.asc \
    && apt-get purge -y --auto-remove curl \
    && gpg --verify mysql.tar.gz.asc \
    && mkdir /usr/local/mysql \
    && tar -xzf mysql.tar.gz -C /usr/local/mysql --strip-compone
```

```
nts=1 \
    && rm mysql.tar.gz* \
    && rm -rf /usr/local/mysql/mysql-test /usr/local/mysql/sql-b
ench \
    && rm -rf /usr/local/mysql/bin/*-debug /usr/local/mysql/bin/
*_embedded \
    && find /usr/local/mysql -type f -name "*.a" -delete \
    && apt-get update && apt-get install -y binutils && rm -rf /
var/lib/apt/lists/* \
    && { find /usr/local/mysql -type f -executable -exec strip -
-strip-all '{}' + || true; } \
    && apt-get purge -y --auto-remove binutils
ENV PATH $PATH:/usr/local/mysql/bin:/usr/local/mysql/scripts

WORKDIR /usr/local/mysql
VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 3306
CMD ["mysqld", "--datadir=/var/lib/mysql", "--user=mysql"]
```

5.7 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assign
ed consistently, regardless of whatever dependencies get added
RUN groupadd -r mysql && useradd -r -g mysql mysql

# FATAL ERROR: please install the following Perl modules before
executing /usr/local/mysql/scripts/mysql_install_db:
# File::Basename
# File::Copy
# Sys::Hostname
# Data::Dumper
RUN apt-get update && apt-get install -y perl --no-install-recom
mends && rm -rf /var/lib/apt/lists/*
```

```
# mysqld: error while loading shared libraries: libaio.so.1: can
not open shared object file: No such file or directory
RUN apt-get update && apt-get install -y libaio1 && rm -rf /var/
lib/apt/lists/*

# gpg: key 5072E1F5: public key "MySQL Release Engineering <mysq
l-build@oss.oracle.com>" imported
RUN gpg --keyserver pgp.mit.edu --recv-keys A4A9406876FCBD3C4567
70C88C718D3B5072E1F5

ENV MYSQL_MAJOR 5.7
ENV MYSQL_VERSION 5.7.4-m14

# note: we're pulling the *.asc file from mysql.he.net instead o
f dev.mysql.com because the official mirror 404s that file for w
hatever reason - maybe it's at a different path?
RUN apt-get update && apt-get install -y curl --no-install-recom
mends && rm -rf /var/lib/apt/lists/* \
    && curl -SL "http://dev.mysql.com/get/Downloads/MySQL-$MYSQL
_MAJOR/mysql-$MYSQL_VERSION-linux-glibc2.5-x86_64.tar.gz" -o mys
ql.tar.gz \
    && curl -SL "http://mysql.he.net/Downloads/MySQL-$MYSQL_MAJO
R/mysql-$MYSQL_VERSION-linux-glibc2.5-x86_64.tar.gz.asc" -o mysq
l.tar.gz.asc \
    && apt-get purge -y --auto-remove curl \
    && gpg --verify mysql.tar.gz.asc \
    && mkdir /usr/local/mysql \
    && tar -xzf mysql.tar.gz -C /usr/local/mysql --strip-compone
nts=1 \
    && rm mysql.tar.gz* \
    && rm -rf /usr/local/mysql/mysql-test /usr/local/mysql/sql-b
ench \
    && rm -rf /usr/local/mysql/bin/*-debug /usr/local/mysql/bin/
*_embedded \
    && find /usr/local/mysql -type f -name "*.a" -delete \
    && apt-get update && apt-get install -y binutils && rm -rf /
var/lib/apt/lists/* \
    && { find /usr/local/mysql -type f -executable -exec strip -
-strip-all '{}' + || true; } \
```

```
    && apt-get purge -y --auto-remove binutils
ENV PATH $PATH:/usr/local/mysql/bin:/usr/local/mysql/scripts

WORKDIR /usr/local/mysql
VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 3306
CMD ["mysqld", "--datadir=/var/lib/mysql", "--user=mysql"]
```

MongoDB

基本信息

MongoDB 是开源的 NoSQL 数据库实现。

该仓库位于 https://hub.docker.com/_/mongo/ ，提供了 MongoDB 2.x ~ 3.x 各个版本的镜像。

使用方法

默认会在 `27017` 端口启动数据库。

```
$ docker run --name some-mongo -d mongo
```

使用其他应用连接到容器，可以用

```
$ docker run --name some-app --link some-mongo:mongo -d applicat
ion-that-uses-mongo
```

或者通过 `mongo`

```
$ docker run -it --link some-mongo:mongo --rm mongo sh -c 'exec
mongo "$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT/tes
t"'
```

Dockerfile

2.6 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assign
ed consistently, regardless of whatever dependencies get added
```

```
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        ca-certificates wget \
        numactl \
    && rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
ENV GOSU_VERSION 1.7
RUN set -x \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/go
su/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architect
ure)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tiano
n/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-archi
tecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B4
2F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/b
in/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true

ENV MONGO_VERSION 2.6.12

RUN set -x \
    && wget -O mongo.tgz "https://fastdl.mongodb.org/linux/mongo
db-linux-x86_64-$MONGO_VERSION.tgz" \
    && wget -O mongo.tgz.sig "https://fastdl.mongodb.org/linux/m
ongodb-linux-x86_64-$MONGO_VERSION.tgz.sig" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys DF
FA3DCF326E302C4787673A01C4E7FAAAB2461C \
    && gpg --batch --verify mongo.tgz.sig mongo.tgz \
    && rm -r "$GNUPGHOME" mongo.tgz.sig \
    && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
    && rm mongo.tgz
```

```
RUN mkdir -p /data/db /data/configdb \
    && chown -R mongodb:mongodb /data/db /data/configdb
VOLUME /data/db /data/configdb

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]
```

3.0 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assign
ed consistently, regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        numactl \
    && rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
ENV GOSU_VERSION 1.7
RUN set -x \
    && apt-get update && apt-get install -y --no-install-recomme
nds ca-certificates wget && rm -rf /var/lib/apt/lists/* \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/go
su/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architect
ure)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tiano
n/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-archi
tecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B4
2F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/b
```

```
in/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apt-get purge -y --auto-remove ca-certificates wget

# gpg: key 7F0CEB10: public key "Richard Kreuter <richard@10gen.
com>" imported
RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-ke
ys 492EAFE8CD016A07919F1D2B9ECBEC467F0CEB10

ENV MONGO_MAJOR 3.0
ENV MONGO_VERSION 3.0.12

RUN echo "deb http://repo.mongodb.org/apt/debian wheezy/mongodb-
org/$MONGO_MAJOR main" > /etc/apt/sources.list.d/mongodb-org.lis
t

RUN set -x \
    && apt-get update \
    && apt-get install -y \
        mongodb-org=$MONGO_VERSION \
        mongodb-org-server=$MONGO_VERSION \
        mongodb-org-shell=$MONGO_VERSION \
        mongodb-org-mongos=$MONGO_VERSION \
        mongodb-org-tools=$MONGO_VERSION \
    && rm -rf /var/lib/apt/lists/* \
    && rm -rf /var/lib/mongodb \
    && mv /etc/mongod.conf /etc/mongod.conf.orig

RUN mkdir -p /data/db /data/configdb \
    && chown -R mongodb:mongodb /data/db /data/configdb
VOLUME /data/db /data/configdb

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]
```

3.2 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assign
ed consistently, regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        numactl \
    && rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
ENV GOSU_VERSION 1.7
RUN set -x \
    && apt-get update && apt-get install -y --no-install-recomme
nds ca-certificates wget && rm -rf /var/lib/apt/lists/* \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/go
su/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architect
ure)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tiano
n/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-archi
tecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B4
2F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/b
in/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apt-get purge -y --auto-remove ca-certificates wget

# pub   4096R/AAB2461C 2014-02-25 [expires: 2016-02-25]
#       Key fingerprint = DFFA 3DCF 326E 302C 4787  673A 01C4 E7
FA AAB2 461C
# uid                 MongoDB 2.6 Release Signing Key <packagin
g@mongodb.com>
#
```

```
# pub   4096R/EA312927 2015-10-09 [expires: 2017-10-08]
#       Key fingerprint = 42F3 E95A 2C4F 0827 9C49  60AD D68F A5
0F EA31 2927
# uid                  MongoDB 3.2 Release Signing Key <packagin
g@mongodb.com>
#
ENV GPG_KEYS \
    DFFA3DCF326E302C4787673A01C4E7FAAAB2461C \
    42F3E95A2C4F08279C4960ADD68FA50FEA312927
RUN set -ex \
    && for key in $GPG_KEYS; do \
        apt-key adv --keyserver ha.pool.sks-keyservers.net --rec
v-keys "$key"; \
    done

ENV MONGO_MAJOR 3.2
ENV MONGO_VERSION 3.2.8

RUN echo "deb http://repo.mongodb.org/apt/debian wheezy/mongodb-
org/$MONGO_MAJOR main" > /etc/apt/sources.list.d/mongodb-org.lis
t

RUN set -x \
    && apt-get update \
    && apt-get install -y \
        mongodb-org=$MONGO_VERSION \
        mongodb-org-server=$MONGO_VERSION \
        mongodb-org-shell=$MONGO_VERSION \
        mongodb-org-mongos=$MONGO_VERSION \
        mongodb-org-tools=$MONGO_VERSION \
    && rm -rf /var/lib/apt/lists/* \
    && rm -rf /var/lib/mongodb \
    && mv /etc/mongod.conf /etc/mongod.conf.orig

RUN mkdir -p /data/db /data/configdb \
    && chown -R mongodb:mongodb /data/db /data/configdb
VOLUME /data/db /data/configdb

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

```
EXPOSE 27017
CMD ["mongod"]
```

3.3 版本

```
FROM debian:jessie

# add our user and group first to make sure their IDs get assign
ed consistently, regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        numactl \
    && rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
ENV GOSU_VERSION 1.7
RUN set -x \
    && apt-get update && apt-get install -y --no-install-recomme
nds ca-certificates wget && rm -rf /var/lib/apt/lists/* \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/go
su/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architect
ure)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tiano
n/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-archi
tecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B4
2F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/b
in/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apt-get purge -y --auto-remove ca-certificates wget

# pub   4096R/A15703C6 2016-01-11 [expires: 2018-01-10]
```

```
#       Key fingerprint = 0C49 F373 0359 A145 1858  5931 BC71 1F
9B A157 03C6
# uid                  MongoDB 3.4 Release Signing Key <packagin
g@mongodb.com>
RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-ke
ys 0C49F3730359A14518585931BC711F9BA15703C6

ENV MONGO_MAJOR 3.3
ENV MONGO_VERSION 3.3.10

RUN echo "deb http://repo.mongodb.org/apt/debian jessie/mongodb-
org/$MONGO_MAJOR main" > /etc/apt/sources.list.d/mongodb-org.lis
t

RUN set -x \
    && apt-get update \
    && apt-get install -y \
        mongodb-org-unstable=$MONGO_VERSION \
        mongodb-org-unstable-server=$MONGO_VERSION \
        mongodb-org-unstable-shell=$MONGO_VERSION \
        mongodb-org-unstable-mongos=$MONGO_VERSION \
        mongodb-org-unstable-tools=$MONGO_VERSION \
    && rm -rf /var/lib/apt/lists/* \
    && rm -rf /var/lib/mongodb \
    && mv /etc/mongod.conf /etc/mongod.conf.orig

RUN mkdir -p /data/db /data/configdb \
    && chown -R mongodb:mongodb /data/db /data/configdb
VOLUME /data/db /data/configdb

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]
```

Redis

基本信息

Redis 是开源的内存 Key-Value 数据库实现。

该仓库位于 https://hub.docker.com/_/redis/ ，提供了 Redis 3.x 各个版本的镜像。

使用方法

默认会在 `6379` 端口启动数据库。

```
$ docker run --name some-redis -d redis
```

另外还可以启用 持久存储。

```
$ docker run --name some-redis -d redis redis-server --appendonl
y yes
```

默认数据存储位置在 `VOLUME/data` 。可以使用 `--volumes-from some-volume-container` 或 `-v /docker/host/dir:/data` 将数据存放到本地。

使用其他应用连接到容器，可以用

```
$ docker run --name some-app --link some-redis:redis -d applicat
ion-that-uses-redis
```

或者通过 `redis-cli`

```
$ docker run -it --link some-redis:redis --rm redis sh -c 'exec
redis-cli -h "$REDIS_PORT_6379_TCP_ADDR" -p "$REDIS_PORT_6379_TC
P_PORT"'
```

Dockerfile

3.0 版本

```
FROM debian:jessie

# add our user and group first to make sure their IDs get assign
ed consistently, regardless of whatever dependencies get added
RUN groupadd -r redis && useradd -r -g redis redis

RUN apt-get update && apt-get install -y --no-install-recommends
 \
        ca-certificates \
        wget \
    && rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
ENV GOSU_VERSION 1.7
RUN set -x \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/go
su/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architect
ure)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tiano
n/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-archi
tecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B4
2F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/b
in/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true

ENV REDIS_VERSION 3.0.7
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3
.0.7.tar.gz
ENV REDIS_DOWNLOAD_SHA1 e56b4b7e033ae8dbf311f9191cf6fdf3ae974d1c

# for redis-sentinel see: http://redis.io/topics/sentinel
RUN buildDeps='gcc libc6-dev make' \
    && set -x \
```

```
    && apt-get update && apt-get install -y $buildDeps --no-inst
all-recommends \
    && rm -rf /var/lib/apt/lists/* \
    && wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL" \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | sha1sum -c -
\
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-component
s=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps

RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data

COPY docker-entrypoint.sh /usr/local/bin/
RUN ln -s usr/local/bin/docker-entrypoint.sh /entrypoint.sh # ba
ckwards compat
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 6379
CMD [ "redis-server" ]
```

3.2 版本

```
FROM debian:jessie

# add our user and group first to make sure their IDs get assign
ed consistently, regardless of whatever dependencies get added
RUN groupadd -r redis && useradd -r -g redis redis

RUN apt-get update && apt-get install -y --no-install-recommends
 \
        ca-certificates \
        wget \
```

```
    && rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
ENV GOSU_VERSION 1.7
RUN set -x \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/go
su/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architect
ure)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tiano
n/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg --print-archi
tecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B4
2F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/b
in/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true

ENV REDIS_VERSION 3.2.3
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3
.2.3.tar.gz
ENV REDIS_DOWNLOAD_SHA1 92d6d93ef2efc91e595c8bf578bf72baff397507

# for redis-sentinel see: http://redis.io/topics/sentinel
RUN buildDeps='gcc libc6-dev make' \
    && set -x \
    && apt-get update && apt-get install -y $buildDeps --no-inst
all-recommends \
    && rm -rf /var/lib/apt/lists/* \
    && wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL" \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | sha1sum -c -
\
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-component
s=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
```

```
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps

RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data

COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 6379
CMD [ "redis-server" ]
```

Nginx

基本信息

Nginx 是开源的高效的 Web 服务器实现，支持 HTTP、HTTPS、SMTP、POP3、IMAP 等协议。

该仓库位于 https://hub.docker.com/_/nginx/ ，提供了 Nginx 1.0 ~ 1.11.x 各个版本的镜像。

使用方法

下面的命令将作为一个静态页面服务器启动。

```
$ docker run --name some-nginx -v /some/content:/usr/share/nginx
/html:ro -d nginx
```

用户也可以不使用这种映射方式，通过利用 Dockerfile 来直接将静态页面内容放到镜像中，内容为

```
FROM nginx
COPY static-html-directory /usr/share/nginx/html
```

之后生成新的镜像，并启动一个容器。

```
$ docker build -t some-content-nginx .
$ docker run --name some-nginx -d some-content-nginx
```

开放端口，并映射到本地的 `8080` 端口。

```
docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

Nginx的默认配置文件路径为 `/etc/nginx/nginx.conf` ，可以通过映射它来使用本地的配置文件，例如

```
docker run --name some-nginx -v /some/nginx.conf:/etc/nginx/ngin
x.conf:ro -d nginx
```

使用配置文件时，为了在容器中正常运行，需要保持 `daemon off;` 。

Dockerfile

stable 版本

```
FROM debian:jessie

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

ENV NGINX_VERSION 1.10.1-1~jessie

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573
BFD6B3D8FBC641079A6ABABF5BD827BD9BF62 \
    && echo "deb http://nginx.org/packages/debian/ jessie nginx"
 >> /etc/apt/sources.list \
    && apt-get update \
    && apt-get install --no-install-recommends --no-install-sugg
ests -y \
                        ca-certificates \
                        nginx=${NGINX_VERSION} \
                        nginx-module-xslt \
                        nginx-module-geoip \
                        nginx-module-image-filter \
                        nginx-module-perl \
                        nginx-module-njs \
                        gettext-base \
    && rm -rf /var/lib/apt/lists/*

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

1.11.x 版本

```
FROM debian:jessie

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

ENV NGINX_VERSION 1.11.1-1~jessie

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573
BFD6B3D8FBC641079A6ABABF5BD827BD9BF62 \
    && echo "deb http://nginx.org/packages/mainline/debian/ jess
ie nginx" >> /etc/apt/sources.list \
    && apt-get update \
    && apt-get install --no-install-recommends --no-install-sugg
ests -y \
                        ca-certificates \
                        nginx=${NGINX_VERSION} \
                        nginx-module-xslt \
                        nginx-module-geoip \
                        nginx-module-image-filter \
                        nginx-module-perl \
                        nginx-module-njs \
                        gettext-base \
    && rm -rf /var/lib/apt/lists/*

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

WordPress

基本信息

WordPress 是开源的 Blog 和内容管理系统框架，它基于 PhP 和 MySQL。

该仓库位于 https://hub.docker.com/_/wordpress/ ，提供了 WordPress 4.x 版本的镜像。

使用方法

启动容器需要 MySQL 的支持，默认端口为 `80` 。

```
$ docker run --name some-wordpress --link some-mysql:mysql -d wo
rdpress
```

启动 WordPress 容器时可以指定的一些环境参数包括：

- `-e WORDPRESS_DB_USER=...` 缺省为 “root”
- `-e WORDPRESS_DB_PASSWORD=...` 缺省为连接 mysql 容器的环境变量 `MYSQL_ROOT_PASSWORD` 的值
- `-e WORDPRESS_DB_NAME=...` 缺省为 “wordpress”
- `-e WORDPRESS_AUTH_KEY=...` , `-e WORDPRESS_SECURE_AUTH_KEY=...` , `-e WORDPRESS_LOGGED_IN_KEY=...` , `-e WORDPRESS_NONCE_KEY=...` , `-e WORDPRESS_AUTH_SALT=...` , `-e WORDPRESS_SECURE_AUTH_SALT=...` , `-e WORDPRESS_LOGGED_IN_SALT=...` , `-e WORDPRESS_NONCE_SALT=...` 缺省为随机 sha1 串

Dockerfile

4.0 版本

```
FROM debian:wheezy

RUN apt-get update && apt-get install -y \
```

```
        apache2 \
        curl \
        libapache2-mod-php5 \
        php5-curl \
        php5-gd \
        php5-mysql \
        rsync \
        wget \
    && rm -rf /var/lib/apt/lists/*
RUN a2enmod rewrite

# copy a few things from apache's init script that it requires t
o be setup
ENV APACHE_CONFDIR /etc/apache2
ENV APACHE_ENVVARS $APACHE_CONFDIR/envvars
# and then a few more from $APACHE_CONFDIR/envvars itself
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_PID_FILE $APACHE_RUN_DIR/apache2.pid
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_LOG_DIR /var/log/apache2
ENV LANG C
RUN mkdir -p $APACHE_RUN_DIR $APACHE_LOCK_DIR $APACHE_LOG_DIR

# make CustomLog (access log) go to stdout instead of files
#  and ErrorLog to stderr
RUN find "$APACHE_CONFDIR" -type f -exec sed -ri ' \
    s!^(\s*CustomLog)\s+\S+!\1 /proc/self/fd/1!g; \
    s!^(\s*ErrorLog)\s+\S+!\1 /proc/self/fd/2!g; \
' '{}' ';'

RUN rm -rf /var/www/html && mkdir /var/www/html
VOLUME /var/www/html
WORKDIR /var/www/html

ENV WORDPRESS_VERSION 4.0.0
ENV WORDPRESS_UPSTREAM_VERSION 4.0

# upstream tarballs include ./wordpress/ so this gives us /usr/s
```

```
rc/wordpress
RUN curl -SL http://wordpress.org/wordpress-${WORDPRESS_UPSTREAM
_VERSION}.tar.gz | tar -xzC /usr/src/

COPY docker-apache.conf /etc/apache2/sites-available/wordpress
RUN a2dissite 000-default && a2ensite wordpress

COPY docker-entrypoint.sh /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]
EXPOSE 80
CMD ["apache2", "-DFOREGROUND"]
```

4.5 版本

```
FROM php:5.6-apache

RUN a2enmod rewrite expires

# install the PHP extensions we need
RUN apt-get update && apt-get install -y libpng12-dev libjpeg-de
v && rm -rf /var/lib/apt/lists/* \
    && docker-php-ext-configure gd --with-png-dir=/usr --with-jp
eg-dir=/usr \
    && docker-php-ext-install gd mysqli opcache

# set recommended PHP.ini settings
# see https://secure.php.net/manual/en/opcache.installation.php
RUN { \
        echo 'opcache.memory_consumption=128'; \
        echo 'opcache.interned_strings_buffer=8'; \
        echo 'opcache.max_accelerated_files=4000'; \
        echo 'opcache.revalidate_freq=60'; \
        echo 'opcache.fast_shutdown=1'; \
        echo 'opcache.enable_cli=1'; \
    } > /usr/local/etc/php/conf.d/opcache-recommended.ini

VOLUME /var/www/html
```

```
ENV WORDPRESS_VERSION 4.5.3
ENV WORDPRESS_SHA1 835b68748dae5a9d31c059313cd0150f03a49269

# upstream tarballs include ./wordpress/ so this gives us /usr/s
rc/wordpress
RUN curl -o wordpress.tar.gz -SL https://wordpress.org/wordpress
-${WORDPRESS_VERSION}.tar.gz \
    && echo "$WORDPRESS_SHA1 *wordpress.tar.gz" | sha1sum -c - \
    && tar -xzf wordpress.tar.gz -C /usr/src/ \
    && rm wordpress.tar.gz \
    && chown -R www-data:www-data /usr/src/wordpress

COPY docker-entrypoint.sh /entrypoint.sh

# grr, ENTRYPOINT resets CMD now
ENTRYPOINT ["/entrypoint.sh"]
CMD ["apache2-foreground"]
```

Node.js

基本信息

Node.js是基于 JavaScript 的可扩展服务端和网络软件开发平台。

该仓库位于 https://hub.docker.com/_/node/ ，提供了 Node.js 0.10 ~ 6.3.x 各个版本的镜像。

使用方法

在项目中创建一个 Dockerfile。

```
FROM node:0.10-onbuild
# replace this with your application's default port
EXPOSE 8888
```

然后创建镜像，并启动容器。

```
$ docker build -t my-nodejs-app
$ docker run -it --rm --name my-running-app my-nodejs-app
```

也可以直接运行一个简单容器。

```
$ docker run -it --rm --name my-running-script -v "$(pwd)":/usr/
src/myapp -w /usr/src/myapp node:0.10 node your-daemon-or-script
.js
```

Dockerfile

0.10 版本

```
FROM buildpack-deps

RUN apt-get update && apt-get install -y \
        ca-certificates \
        curl

# verify gpg and sha256: http://nodejs.org/dist/v0.10.31/SHASUMS
256.txt.asc
# gpg: aka "Timothy J Fontaine (Work) <tj.fontaine@joyent.com>"
RUN gpg --keyserver pgp.mit.edu --recv-keys 7937DFD2AB06298B2293
C3187D33FF9D0246406D

ENV NODE_VERSION 0.10.32

RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/node-v$NODE
_VERSION-linux-x64.tar.gz" \
    && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS2
56.txt.asc" \
    && gpg --verify SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256
.txt.asc | sha256sum -c - \
    && tar -xzf "node-v$NODE_VERSION-linux-x64.tar.gz" -C /usr/l
ocal --strip-components=1 \
    && rm "node-v$NODE_VERSION-linux-x64.tar.gz" SHASUMS256.txt.
asc

CMD [ "node" ]
```

4.x 版本

```
FROM buildpack-deps:jessie

# gpg keys listed at https://github.com/nodejs/node
RUN set -ex \
  && for key in \
    9554F04D7259F04124DE6B476D5A82AC7E37093B \
    94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \
    0034A06D9D9B0064CE8ADF6BF1747F4AD2306D93 \
    FD3A5288F042B6850C66B31F09FE44734EB7990E \
    71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \
    DD8F2338BAE7501E3DD5AC78C273792F7D83545D \
    B9AE9905FFD7803F25714661B63B535A4C206CA9 \
    C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \
  ; do \
    gpg --keyserver ha.pool.sks-keyservers.net --recv-keys "$key
"; \
  done

ENV NPM_CONFIG_LOGLEVEL info
ENV NODE_VERSION 4.4.7

RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-v$NOD
E_VERSION-linux-x64.tar.xz" \
  && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS25
6.txt.asc" \
  && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.tx
t.asc \
  && grep " node-v$NODE_VERSION-linux-x64.tar.xz\$" SHASUMS256.t
xt | sha256sum -c - \
  && tar -xJf "node-v$NODE_VERSION-linux-x64.tar.xz" -C /usr/loc
al --strip-components=1 \
  && rm "node-v$NODE_VERSION-linux-x64.tar.xz" SHASUMS256.txt.as
c SHASUMS256.txt

CMD [ "node" ]
```

5.x 版本

```
FROM buildpack-deps:jessie

# gpg keys listed at https://github.com/nodejs/node
RUN set -ex \
  && for key in \
    9554F04D7259F04124DE6B476D5A82AC7E37093B \
    94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \
    0034A06D9D9B0064CE8ADF6BF1747F4AD2306D93 \
    FD3A5288F042B6850C66B31F09FE44734EB7990E \
    71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \
    DD8F2338BAE7501E3DD5AC78C273792F7D83545D \
    B9AE9905FFD7803F25714661B63B535A4C206CA9 \
    C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \
  ; do \
    gpg --keyserver ha.pool.sks-keyservers.net --recv-keys "$key"; \
  done

ENV NPM_CONFIG_LOGLEVEL info
ENV NODE_VERSION 5.12.0

RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-linux-x64.tar.xz" \
  && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
  && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
  && grep " node-v$NODE_VERSION-linux-x64.tar.xz\$" SHASUMS256.txt | sha256sum -c - \
  && tar -xJf "node-v$NODE_VERSION-linux-x64.tar.xz" -C /usr/local --strip-components=1 \
  && rm "node-v$NODE_VERSION-linux-x64.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt

CMD [ "node" ]
```

6.x 版本

```
FROM buildpack-deps:jessie

# gpg keys listed at https://github.com/nodejs/node
RUN set -ex \
  && for key in \
    9554F04D7259F04124DE6B476D5A82AC7E37093B \
    94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \
    0034A06D9D9B0064CE8ADF6BF1747F4AD2306D93 \
    FD3A5288F042B6850C66B31F09FE44734EB7990E \
    71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \
    DD8F2338BAE7501E3DD5AC78C273792F7D83545D \
    B9AE9905FFD7803F25714661B63B535A4C206CA9 \
    C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \
  ; do \
    gpg --keyserver ha.pool.sks-keyservers.net --recv-keys "$key
"; \
  done

ENV NPM_CONFIG_LOGLEVEL info
ENV NODE_VERSION 6.3.1

RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-v$NOD
E_VERSION-linux-x64.tar.xz" \
  && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS25
6.txt.asc" \
  && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.tx
t.asc \
  && grep " node-v$NODE_VERSION-linux-x64.tar.xz\$" SHASUMS256.t
xt | sha256sum -c - \
  && tar -xJf "node-v$NODE_VERSION-linux-x64.tar.xz" -C /usr/loc
al --strip-components=1 \
  && rm "node-v$NODE_VERSION-linux-x64.tar.xz" SHASUMS256.txt.as
c SHASUMS256.txt

CMD [ "node" ]
```

Docker 命令查询

基本语法

Docker 命令有两大类，客户端命令和服务端命令。前者是主要的操作接口，后者用来启动 Docker daemon。

- 客户端命令：基本命令格式为 `docker [OPTIONS] COMMAND [arg...]` ；
- 服务端命令：基本命令格式为 `docker daemon [OPTIONS]` 。

可以通过 `man docker` 或 `docker help` 来查看这些命令。

客户端命令选项

- --config=""：指定客户端配置文件，默认为 `/.docker` ；
- -D=true|false：是否使用 debug 模式。默认不开启；
- -H, --host=[]：指定命令对应 Docker daemon 的监听接口，可以为 unix 套接字（unix:///path/to/socket），文件句柄（fd://socketfd）或 tcp 套接字（tcp://[host[:port]]），默认为 unix:///var/run/docker.sock；
- -l, --log-level="debug|info|warn|error|fatal"：指定日志输出级别；
- --tls=true|false：是否对 Docker daemon 启用 TLS 安全机制，默认为否；
- --tlscacert= /.docker/ca.pem：TLS CA 签名的可信证书文件路径；
- --tlscert= /.docker/cert.pem：TLS 可信证书文件路径；
- --tlscert= /.docker/key.pem：TLS 密钥文件路径；
- --tlsverify=true|false：启用 TLS 校验，默认为否。

daemon 命令选项

- --api-cors-header=""：CORS 头部域，默认不允许 CORS，要允许任意的跨域访问，可以指定为 “*”；
- --authorization-plugin=""：载入认证的插件；
- -b=""：将容器挂载到一个已存在的网桥上。指定为 'none' 时则禁用容器的网络，与 --bip 选项互斥；
- --bip=""：让动态创建的 docker0 网桥采用给定的 CIDR 地址; 与 -b 选项互

斥；

- --cgroup-parent=""：指定 cgroup 的父组，默认 fs cgroup 驱动为 `/docker` ，systemd cgroup 驱动为 `system.slice` ；
- --cluster-store=""：构成集群（如 Swarm）时，集群键值数据库服务地址；
- --cluster-advertise=""：构成集群时，自身的被访问地址，可以为 `host:port` 或 `interface:port` ；
- --cluster-store-opt=""：构成集群时，键值数据库的配置选项；
- --config-file="/etc/docker/daemon.json"：daemon 配置文件路径；
- --containerd=""：containerd 文件的路径；
- -D, --debug=true|false：是否使用 Debug 模式。缺省为 false；
- --default-gateway=""：容器的 IPv4 网关地址，必须在网桥的子网段内；
- --default-gateway-v6=""：容器的 IPv6 网关地址；
- --default-ulimit=[]：默认的 ulimit 值；
- --disable-legacy-registry=true|false：是否允许访问旧版本的镜像仓库服务器；
- --dns=""：指定容器使用的 DNS 服务器地址；
- --dns-opt=""：DNS 选项；
- --dns-search=[]：DNS 搜索域；
- --exec-opt=[]：运行时的执行选项；
- --exec-root=""：容器执行状态文件的根路径，默认为 `/var/run/docker` ；
- --fixed-cidr=""：限定分配 IPv4 地址范围；
- --fixed-cidr-v6=""：限定分配 IPv6 地址范围；
- -G, --group=""：分配给 unix 套接字的组，默认为 `docker` ；
- -g, --graph=""：Docker 运行时的根路径，默认为 `/var/lib/docker` ；
- -H, --host=[]：指定命令对应 Docker daemon 的监听接口，可以为 unix 套接字（unix:///path/to/socket），文件句柄（fd://socketfd）或 tcp 套接字（tcp://[host[:port]]），默认为 unix:///var/run/docker.sock；
- --icc=true|false：是否启用容器间以及跟 daemon 所在主机的通信。默认为 true。
- --insecure-registry=[]：允许访问给定的非安全仓库服务；
- --ip=""：绑定容器端口时候的默认 IP 地址。缺省为 0.0.0.0；
- --ip-forward=true|false：是否检查启动在 Docker 主机上的启用 IP 转发服务，默认开启。注意关闭该选项将不对系统转发能力进行任何检查修改；
- --ip-masq=true|false：是否进行地址伪装，用于容器访问外部网络，默认开启；
- --iptables=true|false：是否允许 Docker 添加 iptables 规则。缺省为 true；
- --ipv6=true|false：是否启用 IPv6 支持，默认关闭；

- -l, --log-level="debug|info|warn|error|fatal"：指定日志输出级别；
- --label="[]"：添加指定的键值对标注；
- --log-driver="json-file|syslog|journald|gelf|fluentd|awslogs|splunk|etwlogs|gcplogs|none"：指定日志后端驱动，默认为 json-file；
- --log-opt=[]：日志后端的选项；
- --mtu=VALUE：指定容器网络的 mtu；
- -p=""：指定 daemon 的 PID 文件路径。缺省为 `/var/run/docker.pid`；
- --raw-logs：输出原始，未加色彩的日志信息；
- --registry-mirror=://：指定 `docker pull` 时使用的注册服务器镜像地址；
- -s, --storage-driver=""：指定使用给定的存储后端；
- --selinux-enabled=true|false：是否启用 SELinux 支持。缺省值为 false。SELinux 目前尚不支持 overlay 存储驱动；
- --storage-opt=[]：驱动后端选项；
- --tls=true|false：是否对 Docker daemon 启用 TLS 安全机制，默认为否；
- --tlscacert= /.docker/ca.pem：TLS CA 签名的可信证书文件路径；
- --tlscert= /.docker/cert.pem：TLS 可信证书文件路径；
- --tlscert= /.docker/key.pem：TLS 密钥文件路径；
- --tlsverify=true|false：启用 TLS 校验，默认为否；
- --userland-proxy=true|false：是否使用用户态代理来实现容器间和出容器的回环通信，默认为 true；
- --userns-remap=default|uid:gid|user:group|user|uid：指定容器的用户命名空间，默认是创建新的 UID 和 GID 映射到容器内进程。

客户端命令

可以通过 `man docker-COMMAND` 或 `docker help COMMAND` 来查看这些命令的具体用法。

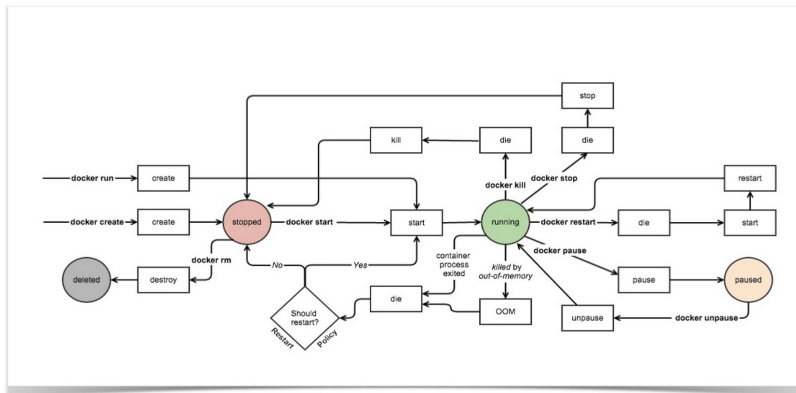
- attach：依附到一个正在运行的容器中；
- build：从一个 Dockerfile 创建一个镜像；
- commit：从一个容器的修改中创建一个新的镜像；
- cp：在容器和本地宿主系统之间复制文件中；
- create：创建一个新容器，但并不运行它；
- diff：检查一个容器内文件系统的修改，包括修改和增加；
- events：从服务端获取实时的事件；

- exec：在运行的容器内执行命令；
- export：导出容器内容为一个 tar 包；
- history：显示一个镜像的历史信息；
- images：列出存在的镜像；
- import：导入一个文件（典型为 tar 包）路径或目录来创建一个本地镜像；
- info：显示一些相关的系统信息；
- inspect：显示一个容器的具体配置信息；
- kill：关闭一个运行中的容器 (包括进程和所有相关资源)；
- load：从一个 tar 包中加载一个镜像；
- login：注册或登录到一个 Docker 的仓库服务器；
- logout：从 Docker 的仓库服务器登出；
- logs：获取容器的 log 信息；
- network：管理 Docker 的网络，包括查看、创建、删除、挂载、卸载等；
- node：管理 swarm 集群中的节点，包括查看、更新、删除、提升/取消管理节点等；
- pause：暂停一个容器中的所有进程；
- port：查找一个 nat 到一个私有网口的公共口；
- ps：列出主机上的容器；
- pull：从一个Docker的仓库服务器下拉一个镜像或仓库；
- push：将一个镜像或者仓库推送到一个 Docker 的注册服务器；
- rename：重命名一个容器；
- restart：重启一个运行中的容器；
- rm：删除给定的若干个容器；
- rmi：删除给定的若干个镜像；
- run：创建一个新容器，并在其中运行给定命令；
- save：保存一个镜像为 tar 包文件；
- search：在 Docker index 中搜索一个镜像；
- service：管理 Docker 所启动的应用服务，包括创建、更新、删除等；
- start：启动一个容器；
- stats：输出（一个或多个）容器的资源使用统计信息；
- stop：终止一个运行中的容器；
- swarm：管理 Docker swarm 集群，包括创建、加入、退出、更新等；
- tag：为一个镜像打标签；
- top：查看一个容器中的正在运行的进程信息；
- unpause：将一个容器内所有的进程从暂停状态中恢复；
- update：更新指定的若干容器的配置信息；

- version：输出 Docker 的版本信息；
- volume：管理 Docker volume，包括查看、创建、删除等；
- wait：阻塞直到一个容器终止，然后输出它的退出符。

一张图总结 Docker 的命令

container 事件状态图



docker 命令分布图

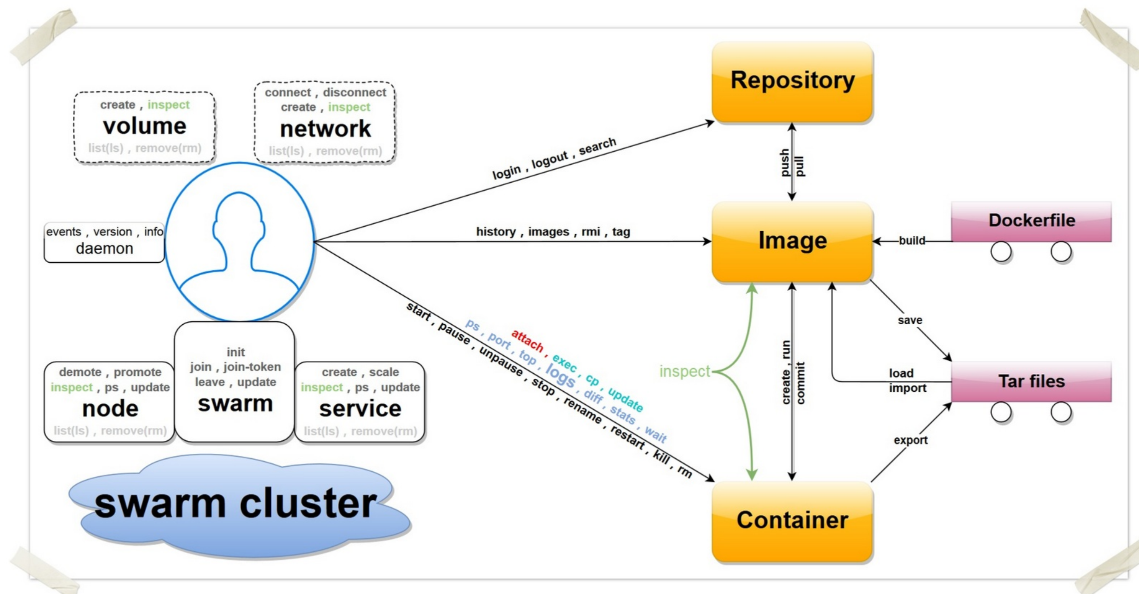


图 *1.24.3.1 - Docker* 命令总结

资源链接

官方网站

- Docker 官方主页: https://www.docker.com
- Docker 官方博客: https://blog.docker.com/
- Docker 官方文档: https://docs.docker.com/
- Docker Hub: https://hub.docker.com
- Docker 的源代码仓库: https://github.com/docker/docker
- Docker 发布版本历史：https://docs.docker.com/release-notes/
- Docker 常见问题：https://docs.docker.com/engine/faq/
- Docker 远端应用 API: https://docs.docker.com/reference/api/docker_remote_api/

实践参考

- Dockerfile 参考：https://docs.docker.com/reference/builder/
- Dockerfile 最佳实践：https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

技术交流

- Docker 邮件列表: https://groups.google.com/forum/#!forum/docker-user
- Docker 的 IRC 频道: https://chat.freenode.net#docker
- Docker 的 Twitter 主页: https://twitter.com/docker

其它

- Docker 的 StackOverflow 问答主页: https://stackoverflow.com/search?q=docker